移动开发系列

Swift 全解析：新式 iOS 实战开发

张云波　冼树华　丁美程　编著

電子工業出版社
Publishing House of Electronics Industry
北京・BEIJING

内 容 简 介

本书从 Swift 4 开始，从 Swift 全面的语法，到 iOS 11 和 iOS 12 的入门级 App 开发，最后由一个非常实用的 iOS 小游戏开发结束。本书是一本适合新手从零开始，一步一步地坚实掌握 iOS 开发技术的一站式启蒙书。本书还配有语法和 iOS 基础开发的视频，可以有效地提升读者的学习效率和理解维度，增强学习兴趣。另外，读者可以与作者无缝交流，快乐地学习 iOS 开发。

本书适合广大编程爱好者及对 iOS 开发感兴趣的读者阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Swift 全解析：新式 iOS 实战开发/张云波，冼树华，丁美程编著. —北京：电子工业出版社，2018.7
（移动开发系列）

ISBN 978-7-121-34610-1

Ⅰ. ①S… Ⅱ. ①张… ②冼… ③丁… Ⅲ. ①程序语言－程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字（2018）第 142617 号

策划编辑：张 迪
责任编辑：底 波
印 刷：
装 订：
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编 100036
开 本：787×1092 1/16 印张：26.25 字数：672 千字
版 次：2018 年 7 月第 1 版
印 次：2018 年 7 月第 1 次印刷
定 价：99.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

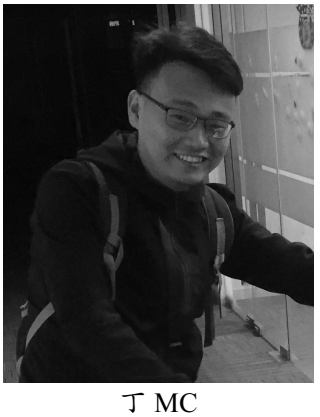
本书咨询联系方式：（010）88254469；zhangdi@phei.com.cn。

前言

借 iOS 每年一次大幅更新的东风，经过几个版本的成长，Swift 4 已经非常成熟了，在 iOS 开发中的便利程度和第三方库的生态建设上，它实际已经大幅领先 Objective-C。苹果所有的官方文档和视频资料均以 Swift 语言为主。这个语言的原作者虽然离开了苹果公司，但 Swift 本身的发展丝毫没有减缓。笔者认为，除了维护垂垂老矣的 Objective-C 项目，新项目完全可以用 Swift 4 从头开始实施。拥抱强类型、类型推断、Optional、面向协议、泛型编程这些现代化编程语言必备的特性，是每个程序员必备的功课之一。

本书分为 3 篇：Swift 4 基础语法篇、iOS 11 基础开发篇和游戏实战篇。若读者在阅读本书时遇到问题，可以进入 QQ 群 532084830，与众多开发者沟通或下载相关资料，也可以在各大视频平台、小波说雨燕官网（xiaoboswift.com）观看本书配套的视频课程，这会使读者的学习效率大幅提升！

作者简介

 小波	上海交通大学毕业，《小波说雨燕》系列 Swift 教学视频作者，表情包式的表演成功吸引了新手的注意。为人勇敢而懦弱，大气却苛刻，长不大的 IT 宅男。人生如戏，常靠临时演技，曾为看动漫而学习日语，后又莫名其妙地获得了东京的工作机会。目前致力于移动 App 领域的开发和培训工作。你有酒，我有故事，直播说给你听。
 华仔	暨南大学珠海学院毕业，IT 大叔一枚，从.Net 到 Java，Android 再到 iOS，从 Winform 到 H5，桌面到 Web，Web 到移动 App，产品研发到数据分析等工作都有涉及，现致力于移动互联网金融相关工作，最近迷上小游戏。
 丁 MC	毕业于安徽工业大学，自学 iOS 开发，参与或独立开发过多款 App 上架 APPStore，目前就职于一家小型软件科技公司任技术总监。一个不甘平凡却依旧平凡的平凡人，相信生活不止眼前的苟且，还有诗和远方。敢于冒险和折腾，正准备着走向创业的不归路。

参与本书编写的还有王倩影、张佳君、王韬懿、陈彬、贾建博、胡佳佳、王艳鹏、郭艾超。

目录

第 1 篇　Swift 4 基础语法篇

第 1 章　关于 Swift …… 3

第 2 章　Swift 语法基础 …… 7

2.1　变量和常量 …… 7
2.2　注释 …… 8
2.3　分号 …… 9
2.4　整数 …… 9
2.5　浮点数 …… 10
2.6　类型安全和类型推断 …… 10
2.7　数值型字面量 …… 10
2.8　数值类型转换 …… 11
2.9　布尔类型 …… 13
2.10　元组 …… 13
2.11　可选类型 …… 14
2.12　错误处理 …… 15
2.13　断言 …… 16

第 3 章　Swift 基本运算符 …… 17

3.1　术语 …… 17
3.2　赋值运算符 …… 17
3.3　算术运算符 …… 18
3.4　组合赋值运算符 …… 19
3.5　比较运算符 …… 19
3.6　三元（目）运算符 …… 20
3.7　空合运算符 …… 21
3.8　区间运算符 …… 22
3.9　逻辑运算符 …… 23

第 4 章　Swift 字符串和字符 …… 26

4.1 字符串字面量……26
4.2 初始化空字符串……26
4.3 字符串可变性……27
4.4 字符串都是值类型……27
4.5 使用字符……27
4.6 连接字符串与字符……28
4.7 字符串插入值……28
4.8 Unicode……29
4.9 计算字符数量……31
4.10 访问与修改字符串……31
4.11 比较字符串……33
第 5 章 Swift 集合类型……35
5.1 数组……35
5.2 集……39
5.3 字典……42
第 6 章 Swift 控制流……46
6.1 for 循环……46
6.1.1 for-in……47
6.1.2 for 条件递增……48
6.2 while 循环……48
6.2.1 while……49
6.2.2 repeat while……49
6.3 if 语句……50
6.4 guard 语句……51
6.5 switch 语句……52
6.6 控制转移语句……54
6.6.1 continue 语句……54
6.6.2 break 语句……55
6.6.3 fallthrough 语句……56
6.6.4 return 语句……57
第 7 章 Swift 函数……58
7.1 函数的定义和调用……58
7.2 函数的参数和返回值……59
7.2.1 多个输入参数的函数……59
7.2.2 没有参数的函数……59

7.2.3 没有返回值的函数……60
7.2.4 没有返回值的函数调用具有返回值的函数……60
7.2.5 多个返回值的函数……60
7.2.6 元组可选类型返回值的函数……61
7.3 函数的参数名称……62
7.3.1 外部参数名称……62
7.3.2 带默认值参数的函数……62
7.3.3 可变参数的函数……63
7.3.4 常量参数和变量参数……63
7.3.5 输入和输出参数……64
7.4 函数类型……65
7.4.1 使用函数类型……65
7.4.2 使用函数类型作为参数……65
7.4.3 使用函数类型作为返回值……66
7.4.4 使用函数类型作为常量……66
7.5 嵌套函数……67
第 8 章 Swift 闭包……68
8.1 闭包的定义……68
8.2 闭包表达式……69
8.2.1 sorted 函数……69
8.2.2 使用闭包表达式……70
8.2.3 根据上下文推断参数……70
8.2.4 单表达式闭包省掉返回关键字……70
8.2.5 参数名称缩写……71
8.2.6 运算符函数……71
8.3 挂尾闭包……71
8.4 闭包的值捕获……72
8.5 闭包是引用类型……73
第 9 章 Swift 枚举……74
9.1 枚举的定义、结构和使用……74
9.2 枚举在 switch 语句中的使用……75
9.3 枚举关联值……75
9.4 枚举原始值……76
第 10 章 Swift 类和结构体……77
10.1 类和结构体的对比……77

10.1.1 类和结构体的定义……78
10.1.2 类和结构体的实例……78
10.1.3 类和结构体的属性访问……78
10.2 结构体是值类型……79
10.3 类是引用类型……79
10.4 如何选择类和结构体……79
10.5 集合类型的赋值和复制……80
第 11 章 Swift 属性……81
11.1 存储属性……81
11.1.1 普通存储属性……81
11.1.2 延迟存储属性……82
11.2 计算属性……82
11.3 属性观察器……84
11.4 全局变量和局部变量……85
11.5 静态属性……85
第 12 章 Swift 方法……86
12.1 实例方法……86
12.1.1 使用类中定义的实例方法……86
12.1.2 使用结构体中定义的实例方法……87
12.1.3 使用枚举中定义的实例方法……88
12.2 静态方法……89
12.2.1 使用类中定义的静态方法……89
12.2.2 使用结构体中定义的静态方法……89
第 13 章 Swift 下标……91
13.1 下标的定义……91
13.2 下标的用法……92
13.3 下标脚本……93
第 14 章 Swift 继承……95
14.1 继承的定义……95
14.2 继承的一般用法……95
14.3 重写父类的方法和属性……96
14.3.1 重写父类的方法……96
14.3.2 重写父类的属性……97
14.4 防止父类的特性被重写……98

第 15 章 Swift 初始化 ……99
15.1 初始化的定义 ……99
15.2 为存储属性设置初始值 ……99
15.3 自定义构造函数 ……100
15.4 默认构造函数 ……101
15.5 值类型的初始化 ……102
15.6 引用类型的初始化 ……103
15.6.1 指定构造函数和便利构造函数 ……104
15.6.2 类的代理初始化调用 ……104
15.6.3 两段式构造过程 ……105
15.6.4 构造函数的继承和重写 ……107
15.6.5 自动构造函数的继承 ……108
15.6.6 指定和便利构造器的实战 ……109
15.7 可失败构造函数 ……111
15.8 必要构造函数 ……116
15.9 通过闭包或方法设置属性的默认值 ……116
第 16 章 Swift 析构 ……119
16.1 析构的原理 ……119
16.2 析构的实战 ……119
第 17 章 Swift 错误处理 ……122
17.1 创建错误类型 ……122
17.2 抛出错误 ……123
17.3 捕获和处理错误 ……124
17.4 不处理错误 ……126
17.5 将错误转为可选类型 ……126
第 18 章 Swift 泛型 ……128
18.1 泛型能解决的问题 ……128
18.2 泛型函数 ……129
18.3 类型参数 ……130
18.4 命名类型参数 ……131
18.5 泛型类型 ……131
18.6 泛型类型的扩展 ……133
18.7 类型约束 ……134
18.8 关联类型 ……136
18.9 条件语句 ……138

第 19 章　Swift 扩展 ……140
19.1　扩展的定义和语法 ……140
19.2　扩展计算属性 ……140
19.3　扩展构造函数 ……141
19.4　扩展方法 ……143
19.5　扩展下标 ……144
19.6　扩展嵌套函数 ……144
第 20 章　Swift 协议 ……146
20.1　协议的定义和语法 ……146
20.2　对属性的规定 ……147
20.3　对方法的规定 ……148
20.4　对变异方法的规定 ……149
20.5　对构造函数的规定 ……150
20.6　协议也是一种类型 ……151
20.7　代理 ……152
20.8　在扩展中添加协议 ……155
20.9　协议类型的集合 ……155
20.10　协议的继承 ……156
20.11　协议类型的组合 ……157
20.12　检查协议的一致性 ……158

第 2 篇　iOS 11 基础开发篇

第 21 章　入门级 ……163
21.1　开发环境配置 ……163
21.2　Playground ……164
21.3　第一个 App ……168
21.4　创建用户交互 ……178
第 22 章　初级 ……181
22.1　Auto Layout ……181
22.2　Stack View ……184
22.3　Size Class ……192
22.4　Table View ……197
22.4.1　原型单元格 ……199
22.4.2　代理模式 ……201
22.4.3　连接数据源 ……203

22.5 自定义单元格 ……204
22.6 单元格交互 ……216
22.7 单元格滑动菜单 ……219
22.8 导航和转场 ……226

第 23 章 中级 ……235

23.1 Model 重构 ……235
23.2 静态单元格 ……241
23.3 导航栏、状态条和动态字体的定制技巧 ……248
23.4 视差动画 ……256
23.5 特效、动画和手势 ……259
23.6 JSON 编/解码 ……271
23.7 录入新数据场景 ……276
23.8 真机部署 ……285
23.9 相册、拍照和录制视频 ……287

第 3 篇 游戏实战篇

第 24 章 卡牌斗兽棋 ……299

24.1 项目介绍与分析 ……299
24.2 项目框架搭建 ……301
24.3 首页和棋盘控制器的搭建 ……306
24.4 相关 model 和走棋逻辑的实现 ……330
24.5 单机双人对战的实现 ……337
24.6 人机对战的实现 ……360
24.7 联网对战的实现 ……371
24.8 联网对战的优化 ……400



第 1 篇



Swift 4 基础语法篇

第 1 章
关于 Swift

Swift 是一门极其出色的编程语言，我们可以使用它编写苹果手机 App、基于 Mac 使用的桌面应用程序、苹果手表应用，甚至服务器端的应用程序等！它是一门安全、快速、交互式的编程语言，是现代先进的编程语言与苹果工程师的智慧和各大开源社区的贡献三者结合发展起来的。Swift 对于之前的版本，编译器和语言都进行了优化升级，其图标如图 1-1 所示。



图 1-1

Swift 是一门很好的编程语言，它兼具工业级别的编程语言质量，也具备脚本语言的编程易学易用的特点。我们使用 Swift 在 Playground 中编写代码，不需要任何编译和运行 App，就能直接看到编写代码后的结果，做到所写即所得。

Swift 采用现代编程模式，定义了大量的常见编程错误：

- 变量在使用前检查是否初始化；
- 数组使用索引前检查是否越界；
- 整数使用前检查是否溢出；
- 可选项检查空值是否已显式处理；
- 内存会自动管理；
- 错误处理允许从失败中恢复。

Swift 代码经过编译和优化，能更充分地使用现代的硬件。Swift 的语法和标准库是根据易于学习和使用这一设计思想指导而创建的。Swift 的安全性和运行速度，使它能够轻松编写出“Hello World”到整个操作系统应用级别的程序。

Swift 使用现代的轻量级的语法，集合强大的类型推断和匹配模式，使一个复杂的想法

通过一个简单、明确的方式表达出来。因此，代码不仅容易编写，也很容易阅读和维护。（代码是写给别人看的，不仅是自己能看懂。）

Swift 已经面世多年（已经到第 4 代了），它会继续发展新的特性和功能。我们对 Swift 充满信心，读者赶紧学习吧！

为了增加读者的自信心，下面开始我们的第一个 Swift 的应用程序。

（1）首先，打开 Xcode（写作的版本是 Xcode 9.2 正式版），选择“Get started with a playground”，如图 1-2 所示。



图 1-2

（2）然后选择“IOS”和“Blank”，如图 1-3 所示。

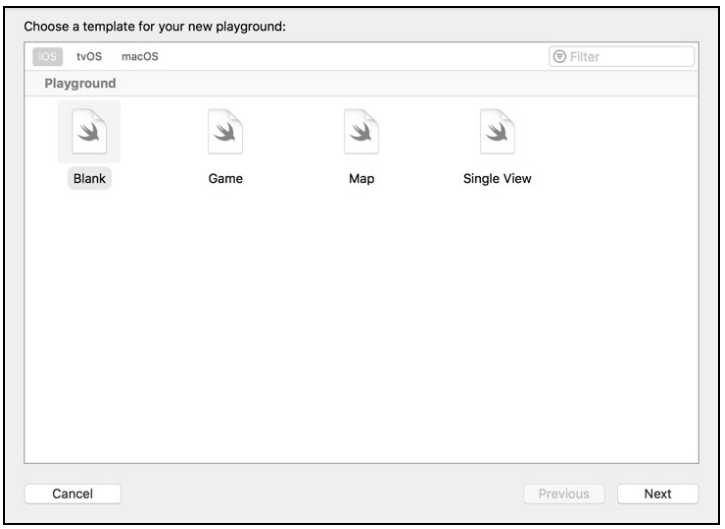


图 1-3

（3）选择好保存的地方，单击“Create”按钮来创建程序，如图 1-4 所示。

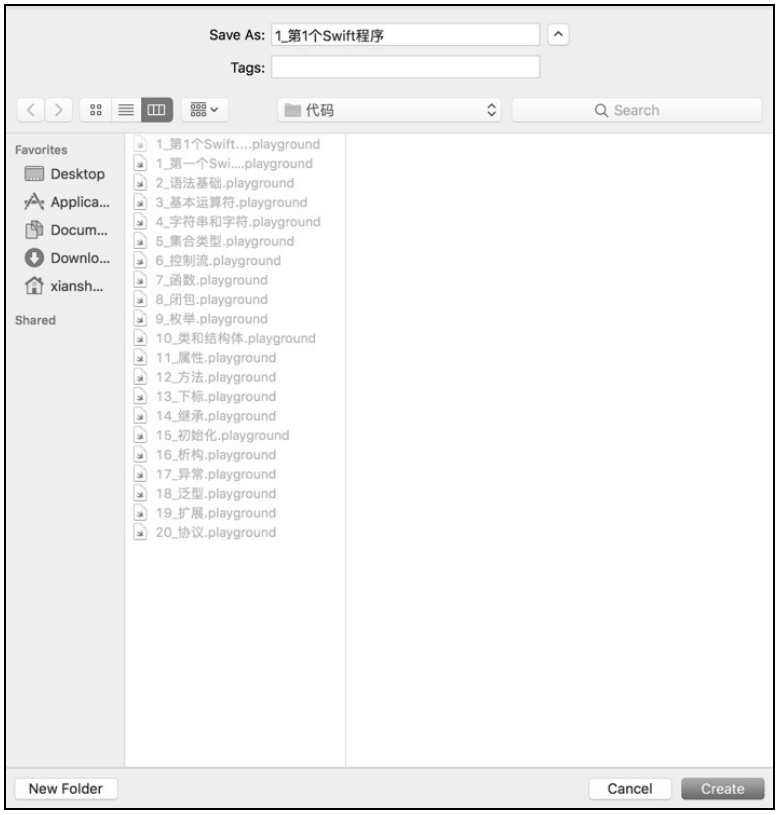


图 1-4

（4）接下来会看到 Xcode 已经帮我们写了一段程序，如图 1-5 所示，图中标注 1 为代码区，标注 2 为显示代码执行结果的结果区



图 1-5

（5）我们来更改一下程序，把代码在隐藏的调试区里打印出来，代码如下。

```
var str = "Hello，大家一起学雨燕"
print(str)
```

（6）打开隐藏的调试区，把打印的信息显示出来，如图 1-6 所示。



图 1-6

简单的 6 个步骤，我们就完成了第一个 Swift 的应用程序，是不是很简洁、很有趣呢？下面我们简单地学习 print()函数。

（1）使用“separator”参数作为多个字符串的分隔符，形成新的字符串打印出来：

```
print("中澳通的同学们", str, "好好学习！", separator: ", ")
```

输出：

中澳通的同学们，Hello， 我们一起学雨燕，好好学习！

（2）使用“terminator”参数作为追加字符串到原有字符串的结尾，形成新的字符串打印出来（\n 代表换行）：

```
print(str, terminator: ", 好吗？\n")
```

输出：

Hello，我们一起学雨燕，好吗？

（3）“separator”参数先使用分隔符连接字符串成为新的字符串，然后再使用“terminator”参数把字符串追加新字符串的结尾：

```
print("志帆教育的", "同学们", str, separator: ", ", terminator: "很有前途！\n")
```

输出：

志帆教育的，同学们，Hello，我们一起学雨燕很有前途！

Swift 的 print()函数就是这么有趣！第 2 章我们开始学习 Swift 语法基础。

第 2 章
Swift 语法基础

本章将简述 Swift 的相关语法基础，包含以下内容。

- 变量和常量
- 注释
- 分号
- 整数
- 浮点数
- 类型安全和类型推断
- 数值型字面量
- 数值类型转换
- 布尔类型
- 元组
- 可选类型
- 错误处理
- 断言

2.1 变量和常量

常量和变量的定义：把一个名字和一个指定类型的值关联起来，常量的值一旦设定就不能改变，变量的值可以随意更改。

如何使用常量和变量呢？我们使用前，必须声明常量和变量，用 let 来声明常量，用 var 来声明变量，代码如下。

```
let MAX_VALUE = 255
var currentCount = 100
```

上面两行代码的意思是：声明一个名字为 MAX_VALUE，并赋值为 255 的常量（常量我们一般习惯使用全部大写英文字母来命名，词语之间使用“_”号是为了更好地阅读）；声明一个名字为 currentCount，并赋值为 100 的变量。

也可以在一行中声明多个常量或变量，使用逗号隔开：

```
let MAX_HEIGHT = 300，MAX_WIDTH = 400，MIN_HEIGHT = 100， MIN_WIDTH = 200
```

```
var x = 30.15， y = 22.8， z = 10.5
```

当声明常量或变量时也可以加上类型标注，注明当前常量或变量的值是什么类型的，很简单，只要在常量或变量名称上加上冒号+空格+类型名称，如：

```
var strResult : String = "你好，变量类型是 String"
let IPADDRESS : String = "http://192.168.1.1"
```

这两行代码的意思是：声明一个 String 类型的变量，名称为 strResult，值是“你好，变量类型是 String”；声明一个 String 类型的常量，名称为 IPADDRESS，值是“http://192.168.1.1”（一般来说，Swift 可以通过赋值来推断出相关数据类型，所以我们给变量和常量添加类型标注的情况不是非常多和必要的）。

变量和常量的命名：不能包含数学符号、箭头、Swift 的关键字、非法的 Unicode 码，连线、制表符；也不能以数字开头，但可以在常量与变量名的其他地方包含数字。

一旦将常量或变量声明为确定的类型，就不能使用相同的名字再次声明或改变它的类型了，同时，也不能将常量与变量进行互相转换。但是，可以使用任何自己喜欢的字符作为常量或变量的名称，甚至可以使用 Unicode 字符：

```
let π = 3.1415926
var say = "你好世界"
var 名字 = "华仔"
let 🐮 = "cow"
```

常量和变量，我们一般使用 print()函数作为输出的方式，打印在调试区里，如上面我们定义好的 4 个常量和变量，可以通过 print()函数输出到调试区里：

```
print(π)
print(say)
print(名字)
print(🐮)
```

输出的结果如图 2-1 所示。



图 2-1

可以看出，print()函数默认添加了\n 这个换行的字符。

2.2 注释

几乎所有的编程语言都必须有，为了方便阅读代码和让人更好地理解代码的意思而提供的一种功能，并且 Swift 的编译器也会在编译代码时自动忽略注释的部分。

和几乎所有的编程语言一样，使用双斜杠（//）来为一行注释，使用单斜杠+星号（/*）作为多行注释的开始，星号+单斜杠（*/）作为多行注释的结束。

注释在 Xcode 的快捷键是 command + /，取消也是。

2.3　分号

Swift 与其他大部分编程语言不同，不会强制要求我们在每条语句的结尾处使用分号（;），当然，我们也可以按照自己的编程习惯来添加。但是，有一种情况下必须使用分号，就是在同一行内写多条独立的代码语句：

```
let cat = "🐱 "; print(cat)
```

上述代码必须使用分号，因为一行内写了两个独立的代码语句，否则编译器会报错。

2.4　整数

整数的定义：没有小数部分的数字，可以是正数、负数和零。

Swift 提供了 8、16、32、64 位的整数类型，我们可以通过访问不同整数类型的 min 和 max 属性获取对应类型的最小和最大值，如图 2-2 所示。

```
Int8.min                                    -128
Int8.max                                    127
Int16.min                                   -32768
Int16.max                                   32767
Int32.min                                   -2147483648
Int32.max                                   2147483647
Int64.min                                   -9223372036854775808
Int64.max                                   9223372036854775807
Int.min                                     -9223372036854775808
Int.max                                     9223372036854775807
```

图 2-2

笔者的机器是 64 位的，可以看到 Int 和 Int64 的最大值和最小值一致，Swift 也推荐我们直接使用 Int，因为 Swift 的 Int 类型会与当前平台的原生字长度一致，Swift 提供的 UInt 与 Int 有类似的功效。

- 在 32 位平台上，Int 和 Int32 的长度相同，UInt 和 UInt32 的长度相同。
- 在 64 位平台上，Int 和 Int64 的长度相同，UInt 和 UInt64 的长度相同。

除非需要特定长度的整数，一般建议使用 Int 即可，这样可以提供代码的一致性和复用性，也尽量不要使用 UInt，除非真的要存储一个正整数数据，除了这种情况，还是直接使用 Int 比较好。尽量避免不同类型数字之间的转换。

2.5 浮点数

浮点数的定义：包含小数部分的数字，例如，3.1415926、0.1、−255.55。

Swift 提供了以下两种浮点数类型。

- Double：64 位浮点数，精度很高，至少 15 位数字。
- Float：32 位浮点数，精度不高，至少 6 位数字。

选择哪种类型的浮点数，取决于我们的需求。

2.6 类型安全和类型推断

Swift 是一个“类型安全”的语言。类型安全的语言可以让我们清楚地知道代码要处理的值的类型。如果我们的代码需要一个 String 类型的值，就绝对不会传过去一个 Int 类型的值过去。

由于 Swift 是“类型安全”的，所以它会在编译代码时进行“类型检查”，并把不匹配的类型标记为错误，这可以让开发者在编译时就能发现 Bug，并进行修复。

然而，也不是要求我们每次声明常量或变量时都需要显式地指定数据类型，因为即使我们没有显式地指定数据类型，Swift 也会使用“类型推断”来选择合适的类型。有了“类型推断”，编译器根据我们给的相关变量、常量或类的属性等的赋值，在编译代码时自动推断表达式的类型（原理：通过赋值推断类型）。

所以，Swift 与其他编程语言相比，很少需要声明类型，只要我们声明常量或变量时赋予它一个“字面量”（值）就会触发“类型推断”，如：

```
let age = 25
```

这行代码会推断 age 是 Int 类型。

```
let userName = "跟华仔学 Swift"
```

这行代码会推断 userName 为 String 类型。

```
let pi = 3.1415
```

这行代码会推断 pi 为 Double 类型。

```
let 圆的面积 = pi * pow(3, 2)
```

这行代码会推断圆的面积也是 Double 类型。

Swift 对于浮点数据的推断，都是推断为精度最高的 Double 类型的。

2.7 数值型字面量

整数字面量可以写作：

- 一个十进制数，没有前缀；

- 一个二进制数，前缀是 0b；
- 一个八进制数，前缀是 0o；
- 一个十六进制数，前缀是 0x。

下面的所有整数字面量的十进制值都是 17:

```
let decimalInteger = 17
let binaryInteger = 0b10001       // 二进制的 17
let octalInteger = 0o21           // 八进制的 17
let hexadecimalInteger = 0x11     // 十六进制的 17
```

浮点字面量可以是十进制（没有前缀）或十六进制（前缀是 0x）。小数点两边必须有至少一个十进制数字（或十六进制数字）。浮点字面量还有一个可选的指数（exponent），在十进制浮点数中通过大写或小写的 e 来指定，在十六进制浮点数中通过大写或小写的 p 来指定。

如果一个十进制数的指数为 exp，那么这个数就相当于基数和 10^exp 的乘积：

- 1.25e2 表示 1.25×10^2，等于 125.0。
- 1.25e−2 表示 1.25×10^−2，等于 0.0125。

如果一个十六进制数的指数为 exp，那么这个数就相当于基数和 2^exp 的乘积：

- 0xFp2 表示 15×2^2，等于 60.0。
- 0xFp−2 表示 15×2^−2，等于 3.75。

下面的这些浮点字面量都等于十进制的 12.1875：

```
let decimalDouble = 12.1875
let exponentDouble = 1.21875e1
let hexadecimalDouble = 0xC.3p0
```

数值类字面量可以包括额外的格式来增强可读性。整数和浮点数都可以添加额外的零并且包含下画线，不会影响字面量：

```
let paddedDouble = 000123.456
let oneMillion = 1_000_000
let justOverOneMillion = 1_000_000.000_000_1
```

2.8　数值类型转换

通常来讲，即使代码中的整数常量和变量已知非负，也请使用“Int”类型。总是使用默认的整数类型可以保证整数常量和变量可以直接被复用，并且可以匹配整数类字面量的类型推断。只有在必要时才使用其他整数类型，如要处理外部长度明确的数据或为了优化性能、内存占用等。使用显式指定长度的类型可以及时发现值溢出，并可以暗示正在处理特殊数据。

不同整数类型的变量和常量可以存储不同范围的数字。“Int8”类型的常量或变量可以存储的数字范围是−128～127，而“UInt8”类型的常量或变量能存储的数字范围是 0～255。

如果数字超出了常量或变量可存储的范围，编译时会报错：

```
let cannotBeNegative: UInt8 = -1
// UInt8 类型不能存储负数，所以会报错
let tooBig: Int8 = Int8.max + 1
// Int8 类型不能存储超过最大值的数，所以会报错
```

由于每种整数类型都可以存储不同范围的值，所以我们必须根据不同的情况选择性地使用数值型类型转换。这种选择性使用的方式，可以预防隐式转换的错误，并让代码中的类型转换意图变得清晰。

要将一种数字类型转换成另一种，要用当前值来初始化一个期望类型的新数字，这个数字的类型就是我们的目标类型。在下面的例子中，常量“twoThousand”是“UInt16”类型，然而常量“one”是“UInt8”类型。因为它们类型不同，不能直接相加，所以要调用“UInt16（one）”来创建一个新的“UInt16”数字并用“one”的值来初始化，然后使用这个新数字来计算：

```
let twoThousand: UInt16 = 2_000
let one: UInt8 = 1
let twoThousandAndOne = twoThousand + UInt16(one)
```

现在两个数字的类型都是“UInt16”，可以进行相加。目标常量“twoThousandAndOne”的类型被推断为“UInt16”，因为它是两个“UInt16”值的和。

“SomeType（ofInitialValue）”是调用 Swift 构造器并传入一个初始值的默认方法。在语言内部，“UInt16”有一个构造器，可以接受一个“UInt8”类型的值，所以这个构造器可以用现有的“UInt8”来创建一个新的“UInt16”。注意，我们并不能传入任意类型的值，只能传入“UInt16”内部有对应构造器的值。不过我们可以扩展现有的类型来让它可以接收其他类型的值（包括自定义类型）。

整数和浮点数的转换必须显式指定类型：

```
let three = 3
let pointOneFourOneFiveNine = 0.14159
let pis = Double(three) + pointOneFourOneFiveNine
// pis 等于 3.14159，所以被推断为 Double 类型
```

在这个例子中，常量“three”的值被用来创建一个“Double”类型的值，所以加号两边的数类型必须相同。如果不进行转换，两者无法相加。

浮点数到整数的反向转换同样可以，整数类型可用 Double 或 Float 类型来初始化：

```
let integerPi = Int(pis)
// integerPi 等于 3，所以被推断为 Int 类型
```

当用这种方式来初始化一个新的整数值时，浮点值会被截断。也就是说，4.75 会变成 4，−3.9 会变成−3。

类型别名（type aliases）就是给现有类型定义另一个名字。可以使用“typealias”关键字来定义类型别名。

当我们想要给现有类型起一个更有意义的名字时，类型别名非常有用。假设我们正在处理特定长度的外部资源的数据：

```
typealias AudioSample = UInt16
```

定义了一个类型别名后，我们可以在任何使用原始名的地方使用别名：

```
var maxAmplitudeFound = AudioSample.min
// maxAmplitudeFound 现在是 0
```

在本例中，“AudioSample”被定义为“UInt16”的一个别名。因为它是别名，“AudioSample.min”实际上是“UInt16.min”，所以会给“maxAmplitudeFound”赋一个初值 0。

2.9　布尔类型

Swift 的布尔类型，定义为 Bool，这个 Bool 只有两种常量的值，要么是 true，要么是 false。布尔类型，一般和条件语句使用比较多，如：

```
let hasKey = true
if(hasKey){
     print("打开门")
}else{
     print("找人开门")
}
```

上面的语句会输出“打开门”。语句的意思是：先是声明了“hasKey”这个布尔类型的常量，然后赋值为“true”。条件语句的意思是：如果“有钥匙”即为真，就输出“打开门”，否则，输出“找人开门”。

2.10　元组

元组的定义：临时把任意数据类型的多个值组合起来的一个复合类型。

一般我们 iOS 手机或手表客户端都是通过 webapi 或 webservice 的形式与服务器交互的，当请求发生错误时都会从服务器返回一个 Http 状态码和一个状态码的说明。例如，请求的路径不存在，就会返回一个 404 状态码和“没找到相关页面”这个状态码的说明。我们可以直接声明其为元组：

```
var http404Error = (404, "没找到相关页面")
```

这里定义了一个“http404Error”的元组，里面包含了一个 Int 类型的值为 404 的状态码和一个 String 类型的值为“没找到相关页面”的状态说明。那么，我们如何使用元组“http404Error”来读取里面状态码和状态说明的数据呢？其实很简单，只要使用下标就可以了（元组的下标都是从 0 开始的），如：

```
//如何获取元组里面的值呢？通过下标，0 是第 1 个元素
var code = http404Error.0
var message = http404Error.1
print("错误代码是：\(code)，错误的信息是：\(message)")
```

```
输出：
错误代码是：404，错误的信息是：没找到相关页面
```

作为函数返回值时，元组非常有用。一个用来获取网页的函数可能会返回一个（Int，String）元组来描述是否获取成功。和只能返回一个类型的值相比，一个包含两个不同类型值的元组可以让函数的返回信息更有用。但是，这并不适合创建复杂的数据结构。如果我们的数据结构并不是临时使用的，应使用类或结构体而不是元组。

2.11 可选类型

可选类型的定义：用来处理可能缺失值的情况。

- 有值：等于值。
- 没值：没有值。

这是否很抽象呢？因为 C、Objective-C、Java 和 C#等语言都没有这个特性，下面我们通过例子来学习，如:

```
let number = "12345"
let isNumber = Int(number)
```

“isNumber”在 Swift 中会被推断为类型“Int?”，或者可选类型“optional Int”，当“number”真的可以转换为 Int 时，“isNumber”的值就是“number”的值，不能转为 Int 时，“isNumber”的值就为 nil。

我们可以使用 if 语句来判断一个可选类型是否包含值，如果可选类型有值，结果为 true；如果没有值，结果为 false（注意，在 print()函数中，可以使用\（变量或常量）作为字符串插入值）。

```
if(isNumber != nil){
     print("可能是数字：\(isNumber)")
}else{
     print("不是数字")
}输出：
可能是数字：Optional(12345)
```

这里我们输出了 Optional（12345）就是可选类型，如何给可选类型拆包呢？很简单，我们使用“!”号就可以给可选类型拆包了，修改一下代码：

```
print("可能是数字：\(isNumber!)")
输出：
可能是数字：12345
```

“isNumber”修改为“isNumber!”就可以给可选类型“Optional”进行强制拆包了。只要我们通过条件语句判断出可选类型是否有值，有值就进行强制拆包，即可使用可选类型里的值了。

我们把上面的语句结合起来，就可以使用可选类型的绑定了，如：

```
if let 是数字么 = Int(number){
      print("是数字么: \(是数字么)")
}else{
      print("不是数字")
}
输出:
是数字么: 12345
```

代码的意思是：如果 Int（number）可以转为 Int 类型，就强行拆包为 Int 类型的数字，并赋值给常量“是数字么”，否则，赋值 nil 给常量“是数字么”；如果“是数字么”不等于 nil，则输出“是数字么 12345”，否则输出“不是数字”。

nil 值：我们只能给可选类型的变量赋予 nil 值，它表示标量没有值。但是，我们不能给非可选类型的常量和变量赋予 nil 值。如果代码中的变量可能是没有值的情况，建议在声明变量时设置为可选变量，它们会被自动设置默认值为 nil，如：

```
var emptyStrng : String?
print(emptyStrng)
输出:
nil
```

如上述代码，可选类型暗示了常量或变量可能“没有值”。我们也可以通过 if 语句来判断是否有值，再通过可选绑定来解析值，但有时我们确定可选类型在第一次被赋非空值后，肯定是会有值的，我们就可以在声明常量或变量时，直接加个感叹号（!）来说明我们这个可选类型是有值的。调用时就不用再判断、拆包或使用可选绑定来解析值了，如：

```
let reward:Int! = 8_000
print(reward)
输出:
8000
```

2.12　错误处理

使用错误处理可处理程序执行中的错误。与可选项做比较，可选项用是否有值来表示一个函数是否执行成功，错误处理允许我们检测程序的失败原因，同时若有必要的话，会把错误传递到另一个执行去处理。

当函数遇到一个错误情况时，它就会“throws”一个错误。函数调用者可以捕获这个错误并适当地处理这个错误。

通过关键字“throws”抛出错误，使用关键字“try”处理抛出的错误，如：

```
func hasError() throws{
    print("处理抛出错误")
}
try hasError()
```

输出：

处理抛出错误

也可以使用“do try catch”的结构捕获错误并进行处理，catch 可以匹配相关错误，不同错误的进行不同的处理，结构如：

```
do {
     try <#throwing expression#>
} catch <#pattern#> {
     <#statements#>
} catch <#pattern#> {
     <#statements#>
}
catch <#pattern#> {
     <#statements#>
}
```

2.13 断言

断言：主要用于在运行时调试，如果条件为 true，则继续执行，如果条件为 false，则终止运行。此外，断言还允许我们附加一条调试信息。如：

```
let andyAge = 16
assert(andyAge>18，"Andy 的年龄没到 18 岁不许喝酒！")
```

输出：

Assertion failed: Andy 的年龄没到 18 岁不许喝酒！

上面代码的意思是：定义了“AndyAge”等于 16 的常量，然后使用断言（assert()函数），如果“AndyAge”大于 18 则程序继续运行，否则，程序停止运行，提示“Andy 的年龄没到 18 岁不许喝酒！”。

第 3 章
Swift 基本运算符

本章将讲述 Swift 的基本运算符，包含以下内容。

- 术语
- 赋值运算符
- 算数运算符
- 组合赋值运算符
- 比较运算符
- 三元（目）运算符
- 空合运算符
- 区间运算符
- 逻辑运算符

3.1 术语

运算符有一元运算符、二元运算符和三元运算符。

- 一元运算符对单一操作对象操作（如−a）。一元运算符分前置运算符和后置运算符，前置运算符需紧排操作对象之前（如!b），后置运算符需紧跟操作对象之后（如 i++）。
- 二元运算符操作两个操作对象（如 2+3），是中置的，因为它们出现在两个操作对象之间。
- 三元运算符操作三个操作对象，和 C 语言一样，Swift 只有一个三元运算符，就是三目运算符（a?b：c）。

受运算符影响的值称操作数，在表达式 1+2 中，“+”是二元运算符，它的两个操作数是值 1 和 2。

3.2 赋值运算符

赋值运算符（a1=b1），表示用 b1 的值来初始化或更新 a1 的值：

```
let b1 = 10
var a1 = 5
```

```
a1 = b1
print("a1 = \(a1)")
输出：
a1 = 10
```

如果赋值的右边是一个多元组，那么它的元素可以马上被分解为多个常量或变量：

```
let (x1, y1) = (1, 2)
print("元组的值 x1 是：\(x1)，y1 的值是：\(y1)")
输出：
元组的值 x1 是：1，y1 的值是：2
```

3.3 算术运算符

Swift 中所有数值类型都支持基本的四则算术运算：

- 加法（+）
- 减法（-）
- 乘法（*）
- 除法（/）

```
let a2 = 10, b2 = 2
print("a2 + b2 = \(a2 + b2)")
print("a2 + b2 = \(a2 - b2)")
print("a2 + b2 = \(a2 * b2)")
print("a2 + b2 = \(a2 / b2)")
print("你的得分是" + "\(a2)分")
输出
a2 + b2 = 12
a2 + b2 = 8
a2 + b2 = 20
a2 + b2 = 5
你的得分是 10 分
```

可以看出，“+”还可以在两个字符串之间起连接作用。

求余运算（a%b）是计算 b 的多少倍刚好可以容入 a，返回多出来的那部分（余数）。

求余运算（%）在其他语言中也称取模运算。然而严格来说，我们看该运算符对负数的操作结果，“求余”比“取模”的说法更合适。

我们来谈谈取余是怎么回事，计算 9%4，先计算出 4 的多少倍会刚好可以容入 9 中：2 倍，那么余数是 1，如图 3-1 所示。

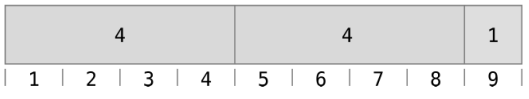


图 3-1

在 Swift 中，我们这样获取余数：

```
let a3 = 9， b3 = 4
print("a3 取余 b3 = \(a3 % b3)")
输出
a3 取余 b3 = 1
```

在对负数−b3 求余时，−b3 的符号会被忽略。这意味着 a3%b3 和 a3%−b3 的结果是相同的，如：

```
let a4 = 9， b4 = -4
print("a4 负数取余 b4 = \(a4 % b4)")
输出
a4 负数取余 b4 = 1
```

数值的正、负号可以使用前缀-（即一元负号）来切换，如：

```
let three = 3
let minusThree = -three      // minusThree 等于 -3
let plusThree = -minusThree   // plusThree 等于 3（因为负负得正）
print("three = \(three)， minusThree = \(minusThree)， plusThree = \(plusThree)")
输出：
three = 3，minusThree = -3，plusThree = 3
```

虽然一元正号什么都不会改变，但当我们在使用一元负号来表达负数时，可以使用一元正号来表达正数，这样的代码会具有对称美。

3.4　组合赋值运算符

如同强大的 C 系列语言（C、C++、Java、C#等），Swift 也提供把其他运算符和赋值运算（=）组合的复合赋值运算符，组合加运算（+=）是其中的一个例子：

```
var a6 = 1
a6 += 2 // a6 现在是 3
print(a6)
输出：
3
```

表达式 a += 2 是 a = a + 2 的简写，一个组合加运算就是把加法运算和赋值运算组合成为一个运算符，同时完成两个运算任务。

3.5　比较运算符

所有标准 C 系列语言中的比较运算都可以在 Swift 中使用。

- 等于（a == b）

- 不等于（a != b）
- 大于（a > b）
- 小于（a < b）
- 大于等于（a >= b）
- 小于等于（a <= b）

注意，Swift 也提供恒等===和不恒等!==这两个比较符来判断两个对象是否引用同一个对象实例。

每个比较运算都返回了一个标识表达式是否成立的布尔值：

- 1 == 1　// true，因为 1 等于 1
- 2 != 1　// true，因为 2 不等于 1
- 2 > 1　// true，因为 2 大于 1
- 1 < 2　// true，因为 1 小于 2
- 1 >= 1　// true，因为 1 大于或等于 1
- 2 <= 1　// false，因为 2 并不小于或等于 1

比较运算多用于条件语句，如 if 条件：

```
let name = "路飞"
if name == "路飞" {
     print("你好，\(name)")
} else {
     print("对不起\(name)，  你不是草帽小子")
}
```

输出：

你好，路飞

上述代码输出“你好，路飞”，是因为“name”等于“路飞”。

3.6　三元（目）运算符

三目运算符的特殊性是它有三个操作数的运算符，它的原型是：问题？答案 1：答案 2。它简洁地表达根据问题成立与否做出二选一的操作。如果问题成立，返回答案 1 的结果；如果不成立，返回答案 2 的结果。

三目运算符是以下代码的缩写形式：

```
if question {
     answer1
} else {
     answer2
}
```

看看下面这个发工资的例子。

```
//工资例子
```

```
let 基本工资 = 8000     //基本工资
let 奖金 = true         //当月是否有奖金
var 实发工资 = 基本工资  //一般情况下实发工资就是基本工作
if(奖金){
     实发工资 = 基本工资 + 5000  //业绩好，发奖金了
}else{
     实发工资 = 基本工资 + 300   //业绩不好，老板随意给点电话补贴
}
print(实发工资)
输出：
13000
```

看上去是否烦琐，我们可以直接把 if else 语句去掉，通过一个三目运算，一个语句就能搞定，修改 if else 的代码，再打印结果：

```
实发工资 = 奖金 ? 基本工资 + 5000 : 基本工资 + 300
print(实发工资)
输出：
13000
```

可以看到，修改后的代码和原来的代码的结果一致，只是三目运算符帮我们简化了 if else 的条件语句。

三目运算提供有效率且便捷的方式来表达二选一的选择。需要注意的事，过度使用三目运算符会使简洁的代码变得难懂。我们应避免在一个组合语句中使用多个三目运算符。

3.7　空合运算符

空合运算符（a ?? b）将对可选类型 a 进行空判断，如果 a 包含一个值就进行解封，否则就返回一个默认值 b。这个运算符有两个条件:

- 表达式 a 必须是 Optional 类型；
- 默认值 b 的类型必须和 a 存储值的类型保持一致。

空合并运算符是对以下代码的简短表达方法：

```
a != nil ? a! : b
```

上述代码使用了三目运算符。当可选类型 a 的值不为空时，强制解封（a!）访问 a 中值，反之当 a 中值为空时，返回默认值 b。无疑空合运算符（??）提供了一种更为优雅的方式去封装条件判断和解封两种行为，显得简洁并更具可读性。

注意，如果 a 为非空值（non-nil），那么值 b 将不会被估值。这也就是所谓的短路求值。

下面的例子采用空合并运算符，实现了在默认颜色名和可选自定义颜色名之间抉择。

```
let defaultColorName = "red"
var userDefinedColorName:String?   //默认值为 nil
var colorNameToUse = userDefinedColorName ?? defaultColorName
```

```
//userDefinedColorName 的值为空 ，所以 colorNameToUse 的值为 red
print("colorNametoUse = \(colorNameToUse)")
输出：
colorNametoUse = red
```

上述的代码“userDefinedColorName”变量被定义为一个可选字符串类型，默认值为nil。由于“userDefinedColorName”是一个可选类型，我们可以使用空合运算符去判断其值。在本例中，通过空合运算符为一个名为“colorNameToUse”的变量赋予一个字符串类型初始值。由于“userDefinedColorName”值为空，因此表达式 userDefinedColorName ?? defaultColorName 返回默认值，即 red。

另一种情况是，分配一个非空值（non-nil）给“userDefinedColorName”，再次执行空合运算，运算结果为封包在“userDefaultColorName”中的值，而非默认值。代码如下：

```
userDefinedColorName = "green"
colorNameToUse = userDefinedColorName ?? defaultColorName
//userDefinedColorName 非空，因此 colorNameToUsede 的值为绿色
print("colorNametoUse = \(colorNameToUse)")
输出：
colorNametoUse = green
```

3.8 区间运算符

Swift 提供了两个方便表达一个区间的值的运算符：

- 闭区间运算符（a...b）定义一个包含从 a 到 b（包括 a 和 b）的所有值的区间，b 必须大于 a。
- 半开区间（a..<b）定义一个从 a 到 b 但不包括 b 的区间。之所以称为半开区间，是因为该区间包含第一个值而不包括最后的值。

闭区间运算符在迭代一个区间的所有值时是非常有用的，如在 for-in 循环中：

```
for index in 1...5 {
    print("\(index) * 5 = \(index * 5)")
}
输出：
1 * 5 = 5
2 * 5 = 10
3 * 5 = 15
4 * 5 = 20
5 * 5 = 25
```

半开区间的实用性在于当我们使用一个 0 始的列表（如数组）时，非常方便地从 0 数到列表的长度。

```
let names = ["小波", "老真", "雨琪", "华仔"]
```

```
let count = names.count
for i in 0..<count {
      print("第 \(i + 1) 个人叫 \(names[i])")
}
输出：
第 1 个人叫 小波
第 2 个人叫 老真
第 3 个人叫 雨琪
第 4 个人叫 华仔
```

数组有 4 个元素，但 0..<count 只数到 3（最后一个元素的下标），因为它是半开区间的。

3.9　逻辑运算符

逻辑运算符的操作对象是逻辑布尔值。Swift 支持基于 C 语言的三个标准逻辑运算。

- 逻辑非（!a）：对一个布尔值取反，使得 true 变 false，false 变 true。
- 逻辑与（a && b）：表达了只有 a 和 b 的值都为 true 时，整个表达式的值才会是 true。
- 逻辑或（a || b）：一个由两个连续的|组成的中置运算符。它表示了两个逻辑表达式的其中一个为 true，整个表达式就为 true。

逻辑非是一个前置运算符，需出现在操作数之前，且不加空格。读作非 a，例子如下：

```
let isHandsome = false
if !isHandsome {
       print("其实华仔很帅！")
}else{
       print("其实华仔不够帅！")
}输出：
其实华仔很帅！
```

“if　！isHandsome”语句可以读作“如果 非 isHandsome。”，接下来的一行代码只有在如果“非 isHandsome”为 true，即“华仔很帅”为 false 时被执行。

在示例代码中，小心地选择布尔常量或变量有助于代码的可读性，并且避免使用双重逻辑非运算，或者混乱的逻辑语句。

逻辑与（a && b）只要任意一个值为 false，整个表达式的值就为 false。事实上，如果第一个值为 false，那么是不去计算第二个值的，因为它已经不可能影响整个表达式的结果了。这被称为“短路计算（short-circuit evaluation）”。

以下例子，只有两个 Bool 值都为 true 时才允许进入：

```
let userName = true
let password = false
if userName && password {
      print("登录成功")
```

```
}else{
      print("登录失败")
}
```

输出：

登录失败

逻辑或（a || b）同逻辑与运算类似，逻辑或也是“短路计算”的，当左端的表达式为 true 时，将不计算右边的表达式，因为它不可能改变整个表达式的值了。

以下示例代码中，第一个布尔值“handsome”为 true，但第二个值“rich”为 false，所以整个表达是 true，于是输出“成为男神!”：

```
let handsome = true
let rich = false
if(handsome || rich){
     print("成为男神! ")
}else{
     print("只能成为屌丝了! ")
}输出：
```

成为男神!

逻辑运算符组合计算：我们可以组合多个逻辑运算来表达一个复合逻辑：

```
let enteredDoorCode = true          //门禁密码
let passedRetinaScan = false        //视网膜扫描
let hasDoorKey = false              //有钥匙
let knowsOverridePassword = true    //知道重置密码
if enteredDoorCode && passedRetinaScan || hasDoorKey || knowsOverridePassword
{
     print("欢迎进入!")
} else {
     print("禁止访问! ")
}
```

输出：

欢迎进入!

这个例子使用了含多个&&和||的复合逻辑。但无论怎样，&&和||始终只能操作两个值。所以这实际是三个简单逻辑连续操作的结果。我们来解读一下。

如果我们输入了正确的密码并通过了视网膜扫描，或者我们有一把有效的钥匙，又或者我们知道紧急情况下重置的密码，我们就能把门打开进入。

前两种情况，我们都不满足，所以前两个简单逻辑的结果是 false，但我们是知道紧急情况下重置的密码的，所以整个复杂表达式的值还是 true。

使用括号来明确优先级：为了一个复杂表达式更容易被读懂，在合适的地方使用括号来明确优先级是很有效的，虽然它并非必要。在上个关于门禁权限的例子中，我们给第一部分加个括号，使用它看起来逻辑更明确：

```
if (enteredDoorCode && passedRetinaScan) || hasDoorKey || knowsOverridePassword {
        print("欢迎进入!")
    } else {
        print("禁止访问！")
}
```

括号使得前两个值被看成整个逻辑表达中独立的一部分。虽然有括号和没括号的输出结果是一样的，但对于读代码的人来说，有括号的代码更清晰。可读性比简洁性更重要，在可以让我们的代码变清晰的地方加个括号吧！

第 4 章
Swift 字符串和字符

本章将讲述 Swift 的字符串和字符，包含以下内容。

- 字符串字面量
- 初始化空字符串
- 字符串可变性
- 字符串都是值类型
- 使用字符
- 连接字符串与字符
- 字符串插入值
- Unicode
- 计算字符数量
- 访问与修改字符串
- 比较字符串

4.1 字符串字面量

字符串字面量是由双引号（""）包裹着的具有固定顺序的文本字符集。

字符串字面量可以用于为常量和变量提供初始值。

```
let str = "Hello, playground"
```

str 常量通过字符串字面量进行初始化，Swift 因此推断该常量为 String 类型。

4.2 初始化空字符串

为了构造一个很长的字符串，可以创建一个空字符串作为初始值。可以将空的字符串字面量赋值给变量，也可以初始化一个新的 String 实例：

```
let emptyString = ""                    // 空字符串字面量
let anotherEmptyString = String()       // 初始化 String 实例
```

可以通过检查其 Boolean 类型的 isEmpty 属性来判断该字符串是否为空：

```
if emptyString.isEmpty {
```

```
        print("什么都没有")
}
输出：
什么都没有
```

4.3　字符串可变性

可以通过将一个特定字符串分配给一个变量来对其进行修改，或者分配给一个常量来保证其不会被修改：

```
//字符串的可变性
var variableString = "Horse"
variableString += " and carriage"
// variableString 现在为 "Horse and carriage"
let constantString = "Highlander"
constantString += " and another Highlander"
// 这会报告一个编译错误 (compile-time error)，即常量不可以被修改
```

4.4　字符串都是值类型

Swift 的字符串类型是值类型。如果我们创建一个新的字符串值，当它传递给一个函数或方法时，或者当它被分配给一个常量或变量时，字符串的值被复制。在这些情况下，现有的字符串值的新副本或被创建，新的副本的值将会被使用，而不是原来版本的值。结构体和枚举类型是值类型。

Swift 中的字符串是否可以修改，仅通过定义的是变量还是常量来决定，实现了多种类型可变性操作的统一。

Swift 的编译器会优化字符串的使用情况，以使复制操作在绝对必要的地方。这意味着，当我们使用字符串作为值类型工作时能获得出色的性能。

4.5　使用字符

Swift 的 String 类型表示特定序列的 Character（字符）类型值的集合。每一个字符值代表一个 Unicode 字符。我们可利用 for-in 循环来遍历字符串中的每一个字符：

```
for character in "Dog! 🐶 " {
     print(character)
}
D
```

```
o
g
!
🐶
```

另外，通过标明一个 Character 类型注解并通过字符字面量进行赋值，可以建立一个独立的字符常量或变量：

```
let rmb: Character = "¥"
print(rmb)
输出：
¥
```

4.6 连接字符串与字符

字符串可以通过加法运算符（+）相加在一起（或称“串联”），并创建一个新的字符串：

```
let string1 = "你好"
let string2 = " 华仔"
var welcome = string1 + string2
print(welcome)
// welcome 现在等于 "你好 华仔"
输出：
你好 华仔
```

也可以通过加法赋值运算符（+=）将一个字符串添加到一个已经存在字符串变量上：

```
var instruction = "我喜欢 "
instruction += string2
print(instruction)
// instruction 现在等于  "我喜欢 华仔"
输出：
我喜欢  华仔
```

还可以用 append 方法将一个字符附加到一个字符串变量的尾部：

```
let exclamationMark: Character = "! "
welcome.append(exclamationMark)
print(welcome)
// welcome 现在等于 "你好 华仔! "
```

注意，不能将一个字符串或字符添加到一个已经存在的字符变量上，因为字符变量只能包含一个字符。

4.7 字符串插入值

字符串插值是一种构建新字符串的方式，可以在其中包含常量、变量、字面量和表达式。插入的字符串字面量的每一项都被包裹在以反斜线为前缀的圆括号中：

```
let multiplier = 3
let message = "\(multiplier) 乘以 2.5 是 \(Double(multiplier) * 2.5)"
print(message)
输出：
3 乘以 2.5 是 7.5
```

在上面的例子中，multiplier 作为\（multiplier）插入到一个字符串字面量中。当创建字符串执行插值计算时，此占位符会被替换为 multiplier 实际的值。

multiplier 的值也作为字符串中后面表达式的一部分。该表达式计算 Double（multiplier）* 2.5 的值，并将结果（7.5）插到字符串中。在这个例子中，表达式写为\（Double（multiplier）* 2.5）并包含在字符串字面量中。

注意，在插值字符串中写在括号内的表达式不能包含非转义双引号（"）和反斜杠（\），并且不能包含回车符或换行符。

4.8　Unicode

Unicode 是一个国际标准，用于文本的编码和表示。它使我们可以用标准格式表示来自任意语言几乎所有的字符，并能够对文本文件或网页这样的外部资源中的字符进行读、写操作。

Swift 的字符串和字符类型是完全兼容 Unicode 标准的，它支持如下所述的一系列不同的 Unicode 编码。

Unicode 中每一个字符都可以被解释为一个或多个 unicode 标量。字符的 unicode 标量是一个唯一的 21 位数字（和名称），例如，U+0061 表示小写的拉丁字母 A（"a"），U+1F425 表示小鸡表情（"🐥"）。

当 Unicode 字符串被写进文本文件或其他存储结构中时，这些 Unicode 标量将会按照 Unicode 定义的集中格式之一进行编码。其包括 UTF-8（以 8 位代码单元进行编码） 和 UTF-16（以 16 位代码单元进行编码）。

Swift 提供了几种不同的方式来访问字符串的 Unicode 表示。

可以利用 for-in 来对字符串进行遍历，从而以 Unicode 字符的方式访问每个字符值。

另外，能够以其他 3 种 Unicode 兼容的方式访问字符串的值：

- UTF-8 代码单元集合（利用字符串的 utf8 属性进行访问）；
- UTF-16 代码单元集合（利用字符串的 utf16 属性进行访问）；
- 21 位的 Unicode 标量值集合（利用字符串的 unicodeScalars 属性进行访问）。

下面由 D、o、g、!和🐶（DOG FACE，Unicode 标量为 U+1F436）组成的字符串中的每个字符代表着每种不同的表示：

```
let dogString = "Dog! 🐶"
```

1. UTF-8

可以通过遍历字符串的 utf8 属性来访问它的 UTF-8 表示。其为 UTF8View 类型的属性，UTF8View 是无符号 8 位（UInt8）值的集合，每个 UInt8 值都是一个字符的 UTF-8 表示：

```
for codeUnit in dogString.utf8{
     print("\(codeUnit) ",false)
}
print("")
输出
68 111 103 33 240 159 144 182
```

在上面的例子中，前四个十进制代码单元值（68，111，103，33）代表了字符 D、o、g 和!，它们的 UTF-8 表示与 ASCII 表示相同。后四个代码单元值（240，159，144，182）是 DOG FACE 的 4 字节 UTF-8 表示。

2. UTF-16

可以通过遍历字符串的 utf16 属性来访问它的 UTF-16 表示。其为 UTF16View 类型的属性，UTF16View 是无符号 16 位（UInt16）值的集合，每个 UInt16 都是一个字符的 UTF-16 表示：

```
for codeUnit in dogString.utf16{
     print("\(codeUnit) ",false)
}
print("")
输出：
68 111 103 33 55357 56374
```

同样，前 4 个代码单元值（68，111，103，33）代表了字符 D、o、g 和!，它们的 UTF-16 代码单元和 UTF-8 完全相同。第 5 个和第 6 个代码单元值（55357 和 56374）是 DOG FACE 字符的 UTF-16 表示。第 1 个值为 U+D83D（十进制值为 55357），第 2 个值为 U+DC36（十进制值为 56374）。

可以通过遍历字符串的 unicodeScalars 属性来访问它的 Unicode 标量表示。其为 UnicodeScalarView 类型的属性，UnicodeScalarView 是 UnicodeScalar 的集合。UnicodeScalar 是 21 位的 Unicode 代码点。

每个 UnicodeScalar 拥有一个值属性，可以返回对应的 21 位数值，用 UInt32 来表示。

```
for scalar in dogString.unicodeScalars {
     print("\(scalar.value) ", false)
}
print("")
输出：
68 111 103 33 128054
```

同样，前 4 个代码单元值（68，111，103，33）代表了字符 D、o、g 和!。第 5 位数值 128054 是一个十六进制 1F436 的十进制表示，其等同于 DOG FACE 的 Unicode 标量 U+1F436。

4.9　计算字符数量

可以使用字符串的“characters.count”属性来统计字符串的字符数量，如：

```
let strResult = "你好，我爱 swift！"
print("常量字符串 strResult 的长度是：\(strResult.count)")
输出：
常量字符串 strResult 的长度是：11
```

4.10　访问与修改字符串

这次，我们主要通过例子来对访问和修改字符串做个说明，读者多输入代码才能更好地理解和使用：

```
//1.字符串的索引
let strs = "Hello, 让华仔来给你讲 Swift 语法,OK"
print(strs.startIndex.encodedOffset)  //strs 的第一个下标
print(strs.endIndex.encodedOffset)    //strs 的最后一个下标
输出
0
24

//2.通过下标获取字符（字符串就是字符的数组，所以下标获取字符，下标都是从 0 开始的）
let a1 = strs[strs.startIndex]                          //第一个下标的字符
let a2 = strs[strs.index(before: strs.endIndex)]    //最后一个字符
print("a1 = \(a1), a2 = \(a2)")
输出：
a1 = H, a2 = K

//3.遍历字符串下标和相关值
for indexs in strs.characters.indices{
   print("\(indexs.encodedOffset),",strs[indexs])
}
输出：
0, H
1, e
2, l
3, l
4, o
```

```
    5, ,
    6,
    7, 让
    8, 华
    9, 仔
    10, 来
    11, 给
    12, 你
    13, 讲
    14, S
    15, w
    16, i
    17, f
    18, t
    19, 语
    20, 法
    21, ,
    22, O
    23, K
    var strHello = "你好，我们学习 Swift"
    //4.插入字符
    strHello.insert("！", at: strHello.endIndex)  //在语句最尾部分插入字符！
    print(strHello)输出：
    你好，我们学习 Swift！

    //5.插入字符串
    strHello.insertContentsOf("你学会了吗？".characters, at: strHello.startIndex)
    //在语句开头插入字符串
    print(strHello)
    输出：
    你学会了吗？你好，我们学习 Swift！

    //6.删除字符，成功就返回删除了的某个字符
    strHello.remove(at: strHello.startIndex)
    print(strHello)
    输出：
    学会了吗？你好，我们学习 Swift！

    //7.删除字符串，成功就返回删除了的某个字符串
    strHello.removeSubrange(strHello.index(strHello.endIndex, offsetBy: -6)..<
strHello.endIndex)
    print(strHello)
```

输出：
学会了吗？你好，我们学习

4.11　比较字符串

Swift 提供了以下 3 种方式来比较字符串的值。

- 字符串相等：两个字符串以同一个顺序包含完全相同的字符，则认为两者字符串相等。
- 前缀相等:字符串是否拥有相同字符的前缀。
- 后缀相等：字符串是否拥有相同字符的后缀。

字符串相等：

```
var a = "华仔"
var b = "华仔"

if(a == b){
      print("相等")
}else{
      print("不相等")
}
```

输出：
相等

通过调用字符串的 hasPrefix/hasSuffix 方法来检查字符串是否拥有特定前缀/后缀。两个方法均需要以字符串作为参数传入并传出 Boolean 值。两个方法均执行基本字符串和前缀/后缀字符串之间逐个字符的比较操作，如：

```
    let placeLists = ["上海市宝山区","上海市徐汇区","上海市普陀区","上海市崇明岛",
"大连市中山区","大连市旅顺口区","大连市小平区","深圳市南山区","上海市福田区","舟山市普陀
区","塔什库尔干吉克自治县","首尔市江南区","鹤岗市南山区","台北市中山区","木垒哈萨克自治
县","博尔塔拉蒙古自治州","巴音郭楞蒙古自治州","伊春市友好区","东京都港区","双鸭山市宝山
区","抚远县黑瞎子岛","玛利亚群岛","舟山群岛","千岛群岛","关岛","南宁市江南区","新疆维吾
尔自治区和田县"]


    var 县数 = 0, 区数 = 0, 自治州数 = 0, 岛数 = 0, 宝山区数 = 0
    var 上海列表 = [String]()
    var 大连列表 = [String]()

    for place in placeLists{
           //通过 hasSuffix 这个后缀相等，获取相关数据统计
           if(place.hasSuffix("县")){
                 ++县数
```

```
            }
            if(place.hasSuffix("区")){
                  ++区数
            }
            if(place.hasSuffix("自治州")){
                  ++自治州数
            }
            if(place.hasSuffix("岛")){
                  ++岛数
            }
            if(place.hasSuffix("宝山区")){
                  ++宝山区数
            }

            //通过 hasPrefix 这个前缀相等，获取相关地方到数组列表里
            if(place.hasPrefix("上海")){
                  上海列表.append(place)
            }
            if(place.hasPrefix("大连")){
                  大连列表.append(place)
            }

    }
    print("一共有\(县数)个县，一共有\(区数)个区，一共有\(自治州数)个自治州，一共有\(岛数)个岛，一共有\(宝山区数)个宝山区")

    for place in 上海列表{
                  print("属于上海的有：\(place)")
    }
    for place in 大连列表{
                  print("属于大连的有：\(place)")
    }
```

输出：

一共有 3 个县，一共有 16 个区，一共有 2 个自治州，一共有 6 个岛，一共有 2 个宝山区
属于上海的有：上海市宝山区
属于上海的有：上海市徐汇区
属于上海的有：上海市普陀区
属于上海的有：上海市崇明岛
属于上海的有：上海市福田区
属于大连的有：大连市中山区
属于大连的有：大连市旅顺口区
属于大连的有：大连市小平区

第 5 章
Swift 集合类型

本章将讲述 Swift 的集合类型，包含以下内容。

- 数组（Array）
- 集（Set）
- 字典（Dictionary）

Swift 语言提供经典的数组（Array）、集（Set）和字典（Dictionary）3 种集合类型来存储集合数据，如图 5-1 所示。

（1）数组（Array）：按顺序存储相同类型的数据。

（2）集（Set）：唯一集的无序集合。

（3）字典（Dictionary）：虽无序存储相同类型数据值，但是需要由独有的标识符引用和寻址（即键值对）。

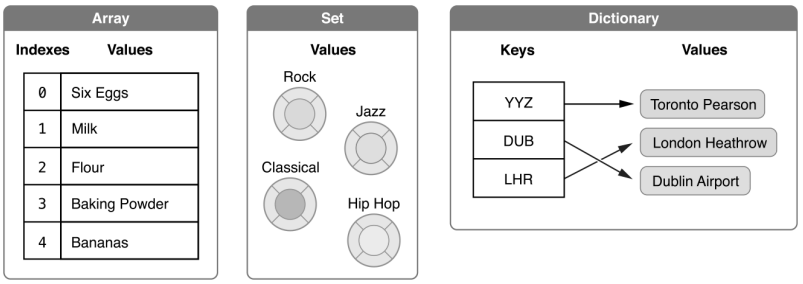


图 5-1

Swift 语言数组（Array）、集（Set）和字典（Dictionary）中存储的数据值类型必须明确。这就意味着，我们不能把不正确的数据类型插入其中，同时这也说明，我们可以更明确从集合里取出的值的类型。Swift 对显式类型集合的使用，确保了我们的代码对工作所需要的类型非常清楚，这也让我们在开发中可以早早地找到任何的类型不匹配错误。

5.1 数组

（1）数组的定义：数组使用有序列表存储同一类型的多个值。相同的值可以多次出现在一个数组的不同位置中。写 Swift 数组应遵循像“Array<SomeType>”这样的形式，其中

“SomeType”是这个数组中唯一允许存在的数据类型。我们也可以使用像“[SomeType]”这样的简单语法。尽管两种形式在功能上是一样的，但 Swift 还是推荐使用较短的那种，而且在本书中都会以这种形式来使用数组。

（2）创建一个数组，我们使用“[SomeType]”来创建一个空数组，如：

```
var peoples = [String]()
print("peoples 是一个字符串类型的数组，\(peoples)")
输出：
peoples 是一个字符串类型的数组，[ ]
```

（3）我们也可以通过数组构造器“count: Int, repeatedValue:String”来创建“count”个带有相同“repeatedValue”值的数组，如：

```
var coders = [String](repeating: "我要加薪", count: 3)
//表示创建了 3 个值都是"我要加薪"的子项（重要的事情说 3 遍）
print("coders: \(coders)")
输出：
coders：["我要加薪", "我要加薪", "我要加薪"]
```

（4）我们甚至可以创建一个数组，它里面的项等于另外两个数组项相加，如：

```
var devDep  = [String](repeating: "代码编写",count: 4)
var testDep = [String](repeating: "产品测试",count: 2)
var myDep = devDep + testDep
print("技术部人员工作：\(myDep)")
输出：
技术部人员工作：["代码编写", "代码编写", "代码编写", "代码编写", "产品测试", "产品测试"]
```

（5）下面，我们创建一个公交车（5 路车）要经过哪些站点的例子，用来说明数组的一些常用属性、判断、方法等操作。

1）我们首先初始化一个数组 bus5，默认经过以下站点：

```
var bus5 = ["香洲总站","南坑","桃园新村","银桦新村","新村","南村","香洲区府"]
print(bus5)
输出：
[香洲总站，南坑，桃园新村，银桦新村，新村，南村，香洲区府]
```

2）我们判断一下 bus5，是否已经默认开通站点了，如果有，统计一下 bus5 要开通的站点数，如：

```
if(bus5.isEmpty){
     print("站点为空")
}
else{
     print("巴士 5 号经过：\(bus5.count)个站")
}
输出：
巴士 5 号经过：7 个站
```

判断是否为空使用了数组的“.isEmpty”属性，统计数组项数量使用了数组的“.count”属性。

3）由于业务需要，bus5 的最后一站，添加一个“红山”站，我们可以使用数组的“append()”方法来实现，如：

```
bus5.append("红山")
print("现在巴士经过的站点有：\(bus5)")
输出：
现在巴士经过的站点有：[香洲总站，南坑，桃园新村，银桦新村，新村，南村，香洲区府，红山]
```

很明显，最后添加了“红山”站。

4）由于“华仔”下周要来“安居园”隔壁的体育中心开 7 场演唱会，公交公司为了方便大家乘坐公共交通工具来看演唱会，临时向原来的第“4”个站点前添加“安居园”站点，我们可以使用数组的“insert（newElement: T, at: Int）”方法，如：

```
bus5.insert("安居园", at:3)
print("现在巴士经过的站点有：\(bus5)")输出：
现在巴士经过的站点有：[香洲总站，南坑，桃园新村，安居园，银桦新村，新村，南村，香洲区府，红山]
```

可以看到，“安居园”已经插入到原来路线的第 4 个站点位置。为什么是“at:3”而不是 4 呢？原因是，数组都是从 0 开始计算的，所以“at:3”就是第 4 位。

5）由于公众需求，要在最后一站“红山”站后添加 4 个站点，我们使用“+=”来实现这一需求，如：

```
bus5 += ["北岭","华侨宾馆","拱北万佳","拱北"]
print("现在巴士经过的站点有：\(bus5)")
输出：
现在巴士经过的站点有：[香洲总站，南坑，桃园新村，安居园，银桦新村，新村，南村，香洲区府，红山，北岭，华侨宾馆，拱北万佳，拱北]
```

6）在所有的公交站点，我们都需要显示公交车经过哪些站点，可以使用 for 循环来实现这个需求，如：

```
for road in bus5{
      print(road, terminator: ", ")
}
print()
输出：
香洲总站 南坑 桃园新村 安居园 银桦新村 新村 南村 香洲区府 红山 北岭 华侨宾馆 拱北万佳 拱北
```

7）由于统计出首站“香洲总站”乘坐人数越来越少，反而“凤凰北”站那边有几个小区，人越来越多，所以，公交公司决定吧首站“香洲总站”修改为“凤凰北”站，我们可以通过数组的下标来实现这一修改，如：

```
bus5[0] = "凤凰北"
print(bus5)
输出：
```

[凤凰北，南坑，桃园新村，安居园，银桦新村，新村，南村，香洲区府，红山，北岭，华侨宾馆，拱北万佳，拱北]

8）再经过乘坐人员的数据分析，公交公司决定把原来的第 5、6、7 站修改为“中山亭”、“前山西门”和“白石”，我们可以通过数组的下标，把这 3 个连续序号的站点一起修改，如：

```
bus5[4...6] = ["中山亭","前山西门","白石"]
print(bus5)
```

输出：

[凤凰北，南坑，桃园新村，安居园，中山亭，前山西门，白石，香洲区府，红山，北岭，华侨宾馆，拱北万佳，拱北]

9）“华仔”演唱会已经完毕，把插入的第 4 站“安居园”站点删除，恢复到之前的路线，我们可以使用数组的“remove（at: Int）”方法实现，如：

```
var delOneStop  = bus5.remove(at: 3)
print("删除站点：\(delOneStop)后，路线是：\(bus5)")
```

输出：

删除站点：安居园后，路线是：[凤凰北，南坑，桃园新村，中山亭，前山西门，白石，香洲区府，红山，北岭，华侨宾馆，拱北万佳，拱北]

10）由于修路问题，我们必须把最后一个站点删除，可以使用数组的“removeLast()”方法去实现，如：

```
var delLastStop = bus5.removeLast()
print("删除最后一个站点：\(delLastStop)")
```

输出：

删除最后一个站点：拱北

11）可以看到，线路和站点经过多次修改，比较混乱，公交公司决定重新在每个站点都显示出最新的站点编号和名称，我们可以使用“for（index, value）in 数组.enumerated()”来实现，其中，“index”就是数组的下标，由于下标都是从 0 开始的，所以站点的编号都是“index + 1”，“value”就是获取数组的值，我们通过数组的“enumerated()”枚举方法来实现，如：

```
for (index,value) in bus5.enumerated(){
      print("第\(index + 1)个站是：\(value)")
}
```

输出：

第 1 个站是：凤凰北
第 2 个站是：南坑
第 3 个站是：桃园新村
第 4 个站是：中山亭
第 5 个站是：前山西门
第 6 个站是：白石
第 7 个站是：香洲区府
第 8 个站是：红山
第 9 个站是：北岭

```
第 10 个站是：华侨宾馆
第 11 个站是：拱北万佳
```

5.2　集

1. 集的定义

集（Set）是指用来存储相同类型并且没有确定顺序的值。当集元素顺序不重要或希望确保每个元素只出现一次时，可以把集合当作数组的另一种形式。Swift 中的 Set 类型被写为 Set<SomeType>，这里的 SomeType 表示 Set 中允许存储的类型，和数组不同的是，集合没有等价的简化形式，我们可以这样创建这个 Set，如：

```
var letters = Set<Character>();
```

然后往里面添加一个元素“a”，如：

```
letters.insert("a")
print(letters)
输出：
["a"]
```

最后，我们清空 Set，可以直接把 Set 赋予空集“[]”，如：

```
letters = []
print(letters)
输出：
[ ]
```

我们以动物园和动物为例，讲述 Set 的创建、比较是否为同一个 Set、判断是否为空、是否包含某一元素和迭代 Set 元素等操作。

1）我们创建两个动物园的 Set，里面包含各种动物，如：

```
var zoon : Set<String> = ["狮子","老虎","斑马","长颈鹿"]
var other : Set = ["狮子","老虎","斑马","长颈鹿"]
```

2）比较两个 Set 是否相等，我们使用“==”号来判断：

```
if(zoon == other){
     print("同一个动物园")
}
输出：
同一个动物园
```

3）判断 Set 是否为空，我们使用 Set 的“.isEmpty”属性，统计 Set 里元素有多少个，我们使用 Set 的“.count”属性，如：

```
if(zoon.isEmpty){
     print("没有动物")
}else{
     print("有很\(zoon.count)种动物")
输出：
```

```
有很 4 种动物
```

4）判断 Set 是否包含某个元素，我们使用 Set 的“contains（member: T）”方法来判断，如：

```
if(zoon.contains("老虎")){
     print("动物园有老虎")
}
输出：
动物园有老虎
```

5）我们可以直接使用 for 循环来迭代输出 Set 里的元素：

```
print("动物园里有：")
for animal in zoon{
     print("\(animal) ", terminator: " ")
}
print("")
输出：
动物园里有：
狮子　斑马　老虎　长颈鹿
```

2. Set 的四大基本操作

（1）交集：使用“intersection”方法，根据两个集合中都包含的值创建的一个新的集合。（A 和 B 共有）

（2）差集：使用“symmericDifference”方法，根据值在一个集合中但不在两个集合中的值创建一个新的集合。（B 中没有 A 的部分）

（3）并集：使用“union”方法，根据两个集合的值创建一个新的集合。（A 合并 B）

（4）补集：使用“subtracting”方法，根据不在该集合中的值创建一个新的集合。（A 没有的合并 B 没有的）

Set 的四大基本操作如图 5-2 所示。

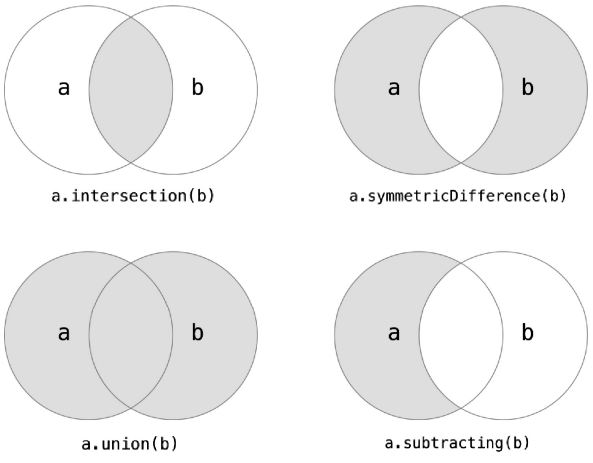


图 5-2

我们使用一个动物园的例子来说明 Set 这 4 种操作。

```
var zoonA : Set<String> = ["狮子","老虎","斑马","长颈鹿"]
var zoonB : Set<String> = ["狮子","牛奶仔","斑马","羊咩仔"]
var result : Set<String> = []
//1.交接:A 和 B 共有
result = zoonA.intersection(zoonB)
print("A 和 B 的交集是: \(result)")
//2.差集: B 中没有 A 的部分
result = zoonA.subtracting(zoonB)
print("A 和 B 的差集是: \(result)")
//3.并集:A 合并 B
result = zoonA.union(zoonB)
print("A 和 B 的并集是: \(result)")
//4.补集:A 没有的合并 B 没有的
result = zoonA.subtracting(zoonB)
print("A 和 B 的补集是: \(result)")
输出:
A 和 B 的交集是: ["斑马", "狮子"]
A 和 B 的差集是: ["牛奶仔", "羊咩仔", "老虎", "长颈鹿"]
A 和 B 的并集是: ["牛奶仔", "羊咩仔", "狮子", "斑马", "老虎", "长颈鹿"]
A 和 B 的补集是: ["老虎", "长颈鹿"]
```

3. Set 的比较操作

图 5-3 所示的插图描述了 3 个集合 a、b 和 c，以及通过悬浮区域表述集合间共享的元素。Set a 是 Set b 的父集合，因为 a 包含了 b 中所有的元素，与之相反，Set b 是 a 的子集合，因为属于 b 的元素也被 a 包含。Set b 和 Set c 彼此不关联，因为它们之间没有共同的元素。

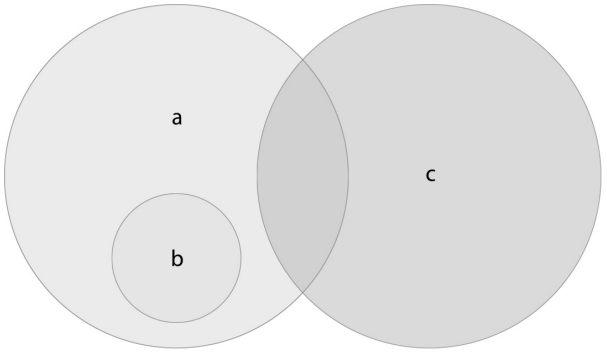


图 5-3

（1）是否相等：使用运算符“==”来判断两个集合是否包含相同的值。

（2）判断子集：使用“isSubsetO”方法来判断一个集合中的值是否也被包含在另外一

个集合中。

（3）判断父集：使用“isSuperset”方法来判断一个集合中包含的值是另一个集合中所有的值。

（4）判断没有相同值：使用“isDisjoint”方法来判断两个结合是否不含有相同的值。

我们再使用一个动物园的例子来对这几种操作进行说明。

```
let zoon1: Set = ["狗", "猫"]
let zoon2: Set = ["牛", "鸡", "羊", "狗", "猫"]
let zoon3: Set = ["乌鸦", "老鼠"]
var res: Bool = false


res = zoon1 == zoon2 ? true : false
print("zoon1 和 zoon2 相等吗？\(res)")

res = zoon1.isSubset(of: zoon2)
print("zoon1 是 zoon2 的子集吗？\(res)")

res = zoon2.isSuperset(of: zoon1)
print("zoon2 是 zoon1 的父集吗？\(res)")

res = zoon2.isDisjoint(with: zoon3)
print("zoon2 和 zoon3 没有相同的元素吗？\(res)")
```

输出：

```
zoon1 和 zoon2 相等吗？false
zoon1 是 zoon2 的子集吗？true
zoon2 是 zoon1 的父集吗？true
zoon2 和 zoon3 没有相同的元素吗？true
```

5.3 字典

1. 字典的定义

字典就是键值对，是一个 key 和一个 value 的结合体。在字典字面量中，每一个键值对的键和值都由冒号分割。这些键值对构成一个列表，其中这些键值对由方括号包含并且由逗号分割。

我们以一个飞机场的例子来说明字典的创建、获取数据、修改数据、判断是否为空、插入数据、删除数据、迭代字典的键和值等操作。

（1）字典的定义，我们可以直接使用“[someType, someType]”的形式来显示定义字典，当然，强大的 Swift 还可以根据我们的赋值去推断出字典的 key 和 value 的类型，如：

```
    var airport1 : [String:String] = ["珠海机场":"123 航班","新加坡机场":"711 航班
","广州机场":"GZ250 航班"]
    var airport2 = ["珠海机场":"123 航班","新加坡机场":"711 航班","广州机场":"GZ250
航班","上海机场":"SH911 航班"]
    print("机场 1：\(airport1)")
    print("机场 2：\(airport2)")
    输出：
    机场 1：["新加坡机场": "711 航班", "广州机场": "GZ250 航班", "珠海机场": "123 航班"]
    机场 2：["新加坡机场": "711 航班", "上海机场": "SH911 航班", "珠海机场": "123 航班",
"广州机场": "GZ250 航班"]
```

（2）我们可以使用字典的“.isEmpty”属性判断字段是否为空，还有“.count”属性来统计字典的元素个数，如：

```
    if(airport1.isEmpty && airport2.isEmpty){
          print("没有航班")
    }else{
          print("机场 1 有航班:\(airport1.count)个,机场 2 有航班:\(airport2.count)
个")
    }
    输出：
    机场 1 有航班：3 个，机场 2 有航班：4 个
```

（3）通过下标插入或更改原有航班，如：

```
    //插入
    airport1["福建机场"] = "FUXXOO"
    //更改数据
    airport2["珠海机场"] = "321 航班"
    print("添加航班的：机场 1：\(airport1)")
    print("更改航班的：机场 2：\(airport2)")
    输出：
    添加航班的：机场 1：["新加坡机场": "711 航班", "珠海机场": "123 航班", "福建机场":
"FUXXOO", "广州机场": "GZ250 航班"]
    更改航班的：机场 2：["新加坡机场": "711 航班", "上海机场": "SH911 航班", "珠海机场":
"321 航班", "广州机场": "GZ250 航班"]
```

（4）使用 key，我们能获取字典里的 value，通过使用“updateValue（value: Value, forKey:key）”方法可以检查 key 是否存在，如果不存在，则返回 nil，如果存在，则返回字典[key]被更新前的值，如：

```
    let oldValue = airport1.updateValue("新 321 航班", forKey: "珠海机场")!
    let newValue = airport1["珠海机场"]!
    print("机场 1 里珠海机场更新前的值是：\(oldValue)，更新后的值是：\(newValue)")
    输出：
    机场 1 里珠海机场更新前的值是：123 航班，更新后的值是：新 321 航班
```

（5）删除值，我们可以使用下标，然后把值设为 nil 或通过“机场 B.removeValue（forKey: Key）”这个方法删除，这个方法是先检查 key 是否存在，存在则删除并返回被删除的旧值，不存在则返回 nil，如：

```
airport1["新加坡机场"] = nil
let re = airport1["新加坡机场"]
print("机场 1 里新加坡机场删除后的值是：\(re)")
输出：
机场 1 里新加坡机场删除后的值是：nil

let delValue = airport2.removeValue(forKey: "新加坡机场")!
let presentValue = airport2["新加坡机场"]
print("机场 2 里新加坡机场删除前的值是：\(delValue)，删除后的值是：\(String(describing: presentValue))")
输出：
机场 2 里新加坡机场删除前的值是：711 航班，删除后的值是：nil
```

6）迭代输出字典的 key、value 和 key value。

```
//迭代 key
for key in airport1.keys{
     print("机场 1 经过的机场有：\(key)")
}
输出：
机场 1 经过的机场有：珠海机场
机场 1 经过的机场有：福建机场
机场 1 经过的机场有：广州机场

//迭代 value
for value in airport1.values{
     print("机场 1 的航班有：\(value)")
}
输出：
机场 1 的航班有：新 321 航班
机场 1 的航班有：FUXXOO
机场 1 的航班有：GZ250 航班

//迭代 key 和 value
for(key,value)in airport1{
     print("机场 1 的航班有：\(key)，经过的机场有：\(value)")
}输出：
机场 1 的航班有：珠海机场，经过的机场有：新 321 航班
机场 1 的航班有：福建机场，经过的机场有：FUXXOO
机场 1 的航班有：广州机场，经过的机场有：GZ250 航班
```

（7）我们甚至可以通过字典 keys 或 values 强制转换，把键和值分别转换为数组。

```
let air = Array(airport2.keys)
print(air)
输出：
["上海机场", "珠海机场", "广州机场"]

let line = [String](airport2.values)
print(line)
输出：
["SH911 航班", "321 航班", "GZ250 航班"]
```

第 6 章
Swift 控制流

本章将讲述 Swift 的控制流，包含以下内容。

- 循环（for、while）
- 条件（if、guard、switch）
- 控制转移（break、continue、fallthrough、return）

Swift 提供了类似 C 语言的流程控制结构，包括可以多次执行任务的 for 和 while 循环，基于特定条件选择执行不同代码分支的 if、guard 和 switch 语句，还有控制流程跳转到其他代码的 break 和 continue 语句。

除了 C 语言里面传统的 for 条件递增（for-condition-increment）循环，Swift 还增加了 for-in 循环，用来更简单地遍历数组（array）、字典（dictionary）、区间（range）、字符串（string）和其他序列类型。

根据特定的条件执行特定的代码通常是十分有用的，例如，当错误发生时，我们可能想运行额外的代码，或者当输入的值太大或太小时，向用户显示一条消息等。要实现这些功能，就需要使用条件语句。

Swift 提供 3 种类型的条件语句：if 语句、guard 语句和 switch 语句。通常，当条件较为简单且可能的情况很少时，使用 if 语句，但是，当我们想一定标配 else 语句的话，就使用 guard 语句。而 switch 语句更适用于条件较复杂、可能情况较多且需要用到模式匹配（pattern-matching）的情形。

Swift 的 switch 语句比 C 语言中的更加强大。在 C 语言中，如果某个 case 不小心漏写了 break，这个 case 就会贯穿（fallthrough）至下一个 case，Swift 无须写 break，所以不会发生这种贯穿（fallthrough）的情况。case 还可以匹配更多的类型模式，包括区间匹配（range matching）、元组（tuple）和特定类型的描述。Switch 的 case 语句中匹配的值可以是由 case 体内部临时的常量或变量决定，也可以由 where 分句描述更复杂的匹配条件。

6.1 for 循环

for 循环用来按照指定的次数多次执行一系列语句。Swift 提供以下两种 for 循环形式。

（1）for-in 用来遍历一个区间（range）、序列（sequence）、集合（collection）、系列（progression）里面所有的元素执行一系列语句。

（2）for 条件递增（for-condition-increment）语句，用来重复执行一系列语句直到达成特定条件，一般通过在每次循环完成后增加计数器的值来实现。

6.1.1　for-in

我们可以使用 for-in 循环来遍历一个集合里面的所有元素，如由数字表示的区间、数组中的元素、字符串中的字符。输出 1～5 的乘法表的代码如下。

```
for index in 1...5{
      print("\(index) 乘以 5 是：\(index * 5)")
}
```

输出：

```
1 乘以 5 是：5
2 乘以 5 是：10
3 乘以 5 是：15
4 乘以 5 是：20
5 乘以 5 是：25
```

本例中用来进行遍历的元素是一组使用闭区间操作符（...）表示的 1～5 的数字。index 被赋值为闭区间中的第一个数字（1），然后循环体中的语句被执行一次。这个循环只包含一个语句，用来输出当前 index 值所对应的乘 5 乘法表结果。该语句执行后，index 的值被更新为闭区间中的第二个数字（2），之后 println 方法会再执行一次。整个过程会进行到闭区间结尾为止。

本例中，index 是一个每次循环遍历开始时被自动赋值的常量。在这种情况下，index 在使用前不需要声明，只要将它包含在循环的声明中，即可对其进行隐式声明，而无须使用 let 关键字进行声明。

如果不需要知道区间内每项的值，我们可以使用下画线（_）替代变量名来忽略对值的访问：

```
let base = 3
let power = 10
var answer = 1

for _ in 1...power{
     answer *= base
}
print("\(base) 的 \(power) 次方是： \(answer)")
```

输出：

```
3 的 10 次方是： 59049
```

这个例子计算 base 这个数的 power 次幂（本例是 3 的 10 次幂），从 1（3 的 0 次幂）开始做 3 的乘法，进行 10 次，使用 1～10 的闭区间循环。这个计算并不需要知道每次循环中计数器具体的值，只要执行了正确的循环次数即可。下画线符号“_”（替代循环中的变量）能够忽略具体的值，并不提供循环遍历时对值的访问。

使用 for-in 循环，我们可以很方便地遍历数组的元素，如：

```
let sports = ["篮球","足球","排球","羽毛球","兵乓球"]
for sport in sports{
    print("我喜欢的运动有：\(sport)")
}
```

输出：

我喜欢的运动有：篮球
我喜欢的运动有：足球
我喜欢的运动有：排球
我喜欢的运动有：羽毛球
我喜欢的运动有：兵乓球

我们也可以通过遍历一个字典来访问它的键值对（key-value pairs）。遍历字典时，字典的每项元素会以（key，value）元组的形式返回，我们可以在 for-in 循环中使用显式的常量名称来解读（key，value）元组。在下面的例子中，字典的键（key）解读为常量“名字”，字典的值会被解读为常量“分数”：

```
let report = ["小明":"80 分","华仔":"99 分","张志":"45 分","苏小小":"68 分","陈胜":"80 分"]
for(name,score) in report{
    print("\(name)：期中考试得分：\(score)")
}
```

输出：

苏小小：期中考试得分：68 分
小明：期中考试得分：80 分
陈胜：期中考试得分：80 分
华仔：期中考试得分：99 分
张志：期中考试得分：45 分

6.1.2 for 条件递增

除了 for-in 循环，Swift 提供了带初始值、限制值和步长的 for 循环:

```
for index in stride(from: 0, to: 3, by: 1) {
      print("index is \(index)")
}
```

输出：

index is 0
index is 1
index is 2

6.2 while 循环

while 循环运行一系列语句直到条件变成 false。这类循环适合使用在第一次迭代前迭代次数未知的情况下。Swift 提供以下两种 while 循环形式。

（1）while 循环形式，每次在循环开始时计算条件是否符合。
（2）repeat-while 循环形式，每次在循环结束时计算条件是否符合。

6.2.1　while

while 循环从计算单一条件开始。如果条件为 true，则会重复运行一系列语句，直到条件变为 false 为止。下面是一般情况下 while 循环格式：

```
while condition { statements }
```

我们来看看使用 while 循环的例子：

```
var start = 0
var finish = 5
while start < finish {
     start++
     print("你好啊")
}
```

输出：

你好啊
你好啊
你好啊
你好啊
你好啊

6.2.2　repeat while

while 循环的另外一种形式是 repeat while，它和 while 的区别是，在判断循环条件之前，先执行一次循环的代码块，然后重复循环直到条件为 false 为止。

下面是一般情况下 do-while 循环的格式：

```
do { statements } while condition
```

我们来看看 repeat while 的例子：

```
var begin = 0
var end = 5
repeat{
    begin++
    print("来跟华仔学习 Swift")
}while begin < end
```

输出：

来跟华仔学习 Swift
来跟华仔学习 Swift
来跟华仔学习 Swift
来跟华仔学习 Swift
来跟华仔学习 Swift

6.3 if 语句

if 语句最简单的形式就是只包含一个条件，当且仅当该条件为 true 时，才执行相关代码，以下是单个条件的例子，如：

```
let temperature = 10
if temperature == 10{
    print("还行啊，一点点冷")
}
```

输出：

还行啊，一点点冷

上面的例子会判断温度是否等于 10℃。如果是，则打印一条消息；否则，不打印任何消息，继续执行 if 块后面的代码。

当然，if 语句允许二选一，也就是当条件为 false 时，执行 else 语句：

```
if temperature > 11{
      print("不算很冷")
}else {
      print("还是很冷呢")
}
```

输出：

还是很冷呢

显然，这两条分支语句中总有一条会被执行。温度小于 11℃，还是很冷呢，因此，else 分支就被触发了。

我们可以把多个 if 语句连接在一起，如：

```
if temperature > 11{
     print("不算很冷")
} else if temperature < 10{
     print("还是很冷呢")
}else {
     print("还行啊，一点点冷")
}
```

输出：

还行啊，一点点冷

在上面的例子中，额外的 if 语句用于判断是不是很冷，而最后的 else 语句被保留了下来，用于打印“一点点冷”时的消息。

6.4　guard 语句

guard 语句和 if 语句类似，都是根据其关键字之后的表达式的布尔值决定下一步执行什么。但它与 if 语句不同的是，guard 语句只会有一个代码块，不像 if 语句可以 if else 多个代码块。

那么，guard 语句的作用到底是什么呢？顾名思义，就是守护。guard 语句判断其后的表达式布尔值为 false 时，才会执行之后代码块里的代码，如果为 true，则跳过整个 guard 语句。

我们以用户登录系统为例，在用户登录系统时会检查账号和密码，如：

```
unc login(person: [String: String]) {
    guard let name = person["name"] else {
        print("请输入账号！")
        return
    }

    print("你好 \(name)!",terminator: "")//不换行

    guard let passwd = person["password"] else {
        print("密码不正确不能登录系统！")
        return
    }

    print("正在登录系统，用户名是:\(name)，密码是:\(passwd)")
}
login(person: ["name":"漩涡鸣人"])
login(person: ["name":"草帽小子","password":"12345"])
输出：
你好 漩涡名人!密码不正确不能登录系统！
你好 草帽小子!正在登录系统，用户名是:草帽小子，密码是:12345
```

上述代码中的第一个 guard 语句用于检查账号，如果检查到没输入账号，也就是表达式为 false 时，执行花括号里的代码，并返回。第二个 guard 语句则检查密码。如果账号和密码正确，则执行最后一个打印语句，上面的两个 guard 语句花括号内的代码都不会执行，因为它们表达式的布尔值都是 true。

guard 语句正如一个称职的守卫，层层把关，严防一切不允许发生的事情，并且让代码具有更高的可读性。

6.5 switch 语句

switch 语句会尝试把某个值与若干个模式（pattern）进行匹配。根据第一个匹配成功的模式，switch 语句会执行对应的代码。当有较多可能的情况时，通常用 switch 语句替换 if 语句。

switch 语句最简单的形式就是把某个值与一个或若干个相同类型的值进行比较：

```
switch some value to consider {
    case value 1: respond to value 1
    case value 2, value 3: respond to value 2 or 3
    default: otherwise, do something else
}
```

switch 语句都由多个 case 构成。为了匹配某些更特定的值，Swift 提供了几种更复杂的匹配模式，这些模式将在本节的后面讲述。

每个 case 都是代码执行的一条分支，这与 if 语句类似。与之不同的是，switch 语句会决定哪一条分支应该被执行。

switch 语句必须是完备的。这就是说，每一个可能的值都必须至少有一个 case 分支与之对应。在某些不可能涵盖所有值的情况下，可以使用默认（default）分支满足该要求，这个默认分支必须在 switch 语句的最后面。

下面的例子使用 switch 语句来匹配一个名“animal”的“猴子”：

```
let animal = "猴子"
switch animal {
case "猪","猴子","羊": print("你是\(animal)")  //不用加 break
case "蛇","老鼠","鸡": print("你是\(animal)")
    //必须加默认值，否则会报错
default: print("默认你是\(animal)")
}
输出：
你是猴子
```

在这个例子中，第一个 case 分支用于匹配三个动物，第二个 case 分支用于匹配另外三个动物，由于为其他可能的 case 分支没有实际意义，因此这个例子使用了默认分支来处理剩其他动物的可能性——这就保证了 switch 语句的完备性。

case 分支的模式也可以是一个值的区间。下面的例子展示了如何使用区间匹配来输出任意数字对应的自然语言格式。

```
let yearIncome = 150_000
switch yearIncome{
case 30_000...60_000:print("码奴：工作一两年，租房混日子")
case 60_001...100_000: print("码农：工作三四年，租房，继续混日子")
case 100_001...200_000: print("IT 民工：经理级别，基本无房，学会装蒜，生活压力大
```

```
")
    case 200_001...300_000: print("IT 工程师：高级经理级别，有房贷，生活压力较大");
    case 300_001...500_000: print("IT 人才：各 IT 公司总监级别人物，有房有车，生活压力相对较小")
    case 500_001...1000000: print("IT 精英：各 IT 公司副总裁级别人物，包括 COO、CTO 等，大多为职业经理人，赚够钱就跑。")
    case 1_000_001...10_000_000: print("IT 大哥：不缺钱，行业里的羡慕嫉妒恨")
    case 10_000_001...100_000_000: print("IT 领袖：任正非、马化腾、李彦宏、丁磊、马云等，包括期权股票以及投资理财等收入")
    default:print("码畜：刚毕业，租房傻乐")
    }
    输出：
    IT 民工：经理级别，基本无房，学会装蒜，生活压力大
```

我们可以使用元组在同一个 switch 语句中测试多个值，如：

```
    var guess = ("大",5)
    switch guess{
    case ("大",6):print("开大，点数是 6")
    case ("大",5):print("开大，点数是 5")
    case ("大",4):print("开大，点数是 4")
    case ("小",3):print("开大，点数是 3")
    case ("小",2):print("开大，点数是 2")
    case ("小",1):print("开大，点数是 1")
    default:("没点，通杀")
    }
    输出：
    开大，点数是 5
```

当然，我们也可以使用元组的值绑定后，再进行比较，如：

```
    guess = ("小",3)
    switch guess{
    case (let x, 6):print("开\(x)，点数是 6")
    case ("大",let y):print("开大，点数是\(y)")
    case let(x,y):print("开\(x)，点数是\(y)")
    }
    输出：
    开小，点数是 3
```

甚至，我们可以使用 where 关键字作为元组值的条件来进行比较，如：

```
    guess = ("通杀",0)
    switch guess{
    case let(x,y) where y>3 : print("开\(x)，点数是：\(y)")
    case let(x,y) where 0<y : print("开\(x)，点数是：\(y)")
    default :print("没点，通杀")
```

```
}
输出：
没点，通杀
```

6.6 控制转移语句

控制转移语句：改变代码的执行顺序，通过它我们可以实现代码的跳转。Swift 有以下四种控制转移语句。

- continue
- break
- fallthrough
- return

6.6.1 continue 语句

continue 语句告诉一个循环体立刻停止本次循环迭代，重新开始下次循环迭代。就好像是说“本次循环迭代我已经执行完了”，但并不会离开整个循环体。

注意，在一个 for 条件递增（for-condition-increment）循环体中，在调用 continue 语句后，迭代增量仍然会被计算求值。循环体继续像往常一样工作，仅是循环体中的执行代码会被跳过。

下面的例子把一个小写字符串中的元音字母和空格字符移除，生成了一个含义模糊的短句：

```
let puzzleInput = "we are study swift"
var puzzleOutput = ""
for character in puzzleInput {
     switch character {
     case "a", "e", "i", "o", "u", " ":
          continue
     default:
          puzzleOutput.append(character)
    }
}
print(puzzleOutput)
输出：
wrstdyswft
```

在上面的代码中，只要匹配到元音字母或空格字符，就调用 continue 语句，使本次循环迭代结束，重新开始下次循环迭代。这种行为使 switch 匹配到元音字母和空格字符时不做处理，而不是让每一个匹配到的字符都被打印。

6.6.2　break 语句

break 语句会立刻结束整个控制流的执行。当我们想要更早地结束一个 switch 代码块或一个循环体时，可以使用 break 语句。

当在一个循环体中使用 break 语句时，会立刻中断该循环体的执行，然后跳转到表示循环体结束的花括号（}）后的第一行代码。不会再有本次循环迭代的代码被执行，也不会再有下次的循环迭代产生。

当在一个 switch 代码块中使用 break 时，会立即中断该 switch 代码块的执行，并且跳转到表示 switch 代码块结束的圆括号后的第一行代码。

这种特性可以被用来匹配或忽略一个或多个分支。因为 Swift 的 switch 代码块需要包含所有的分支且不允许有为空的分支，有时为了使我们的意图更明显，需要特意匹配或忽略某个分支。那么当我们想忽略某个分支时，可以在该分支内写上 break 语句。当那个分支被匹配到时，分支内的 break 语句立即结束 switch 代码块。

注意，当一个 switch 分支仅包含注释时，会被报编译时错误。注释不是代码语句且也不能让 switch 分支达到被忽略的效果。我们总是可以使用 break 来忽略某个分支。

下面的例子通过 switch 来判断一个 Character 值是否代表下面的四种语言之一。为了简洁，多个值被包含在了同一个分支情况中。

```
let numberSymbol: Character = "三"
var possibleIntegerValue: Int?
switch numberSymbol {
case "1", "١", "一", "๑":
    possibleIntegerValue = 1
case "2", "٢", "二", "๒":
    possibleIntegerValue = 2
case "3", "٣", "三", "๓":
    possibleIntegerValue = 3
case "4", "٤", "四", "๔":
    possibleIntegerValue = 4
default:
    break
}
if let integerValue = possibleIntegerValue {
     print("The integer value of \(numberSymbol) is \(integerValue).")
} else {
     print("An integer value could not be found for \(numberSymbol).")
}
```

输出：

```
The integer value of 三 is 3.
```

这个例子检查 numberSymbol 是否是拉丁文、阿拉伯文、中文或泰文中的 1～4 之一。

如果被匹配到，该 switch 分支语句给 Int?类型变量 possibleIntegerValue 设置一个整数值。

当 switch 代码块执行完毕后，接下来的代码通过使用可选绑定来判断 possibleIntegerValue 是否曾经被设置过值。因为是可选类型的缘故，possibleIntegerValue 有一个隐式的初始值 nil，所以仅当 possibleIntegerValue 曾被 switch 代码块的前四个分支中的某个设置过一个值时，可选的绑定将会被判定为成功。

在上面的例子中，想要把 Character 所有的可能性都枚举出来是不现实的，所以使用 default 分支来包含所有上面没有匹配到字符的情况。由于这个 default 分支不需要执行任何动作，所以它只写了一条 break 语句。一旦落到 default 分支中后，break 语句就完成了该分支的所有代码操作，代码继续向下运行，开始执行 if let 语句。

6.6.3 fallthrough 语句

Swift 中的 switch 代码块不会从上一个 case 分支落到下一个 case 分支中。相反，只要第一个匹配到的 case 分支完成了它需要执行的语句，整个 switch 代码块就完成了它的执行操作。相比之下，C 语言要求我们显示地插入 break 语句到每个 switch 分支的末尾来阻止自动落到下一个 case 分支中。Swift 的这种避免默认落到下一个分支中的特性意味着它的 switch 功能要比 C 语言的更加清晰和可预测，可以避免无意识地执行多个 case 分支而引发的错误。

如果我们确实需要 C 风格的贯穿（fallthrough）的特性，可以在每个需要该特性的 case 分支中使用 fallthrough 关键字。下面的例子是使用 fallthrough 关键字来创建一个数字的描述语句。

```
let integerToDescribe = 5
var description = "数字 \(integerToDescribe) 是"
switch integerToDescribe {
case 2, 3, 5, 7, 11, 13, 17, 19:
    description += " 质数，并且是"
    fallthrough
default:
    description += " 一个整数。"
}
print(description)
输出：
数字 5 是 质数，并且是 一个整数。
```

这个例子定义了一个 String 类型的变量 description，并给它设置了一个初始值。函数使用 switch 逻辑来判断 integerToDescribe 变量的值。当 integerToDescribe 的值属于列表中的质数之一时，该函数添加一段文字在 description 后，来表明这个是数字并且是一个质数。然后它使用 fallthrough 关键字来“贯穿”到 default 分支中。default 分支添加一段额外的文字在 description 的最后面，至此，switch 代码块执行完毕。

如果 integerToDescribe 的值不属于列表中的任何质数，那么它不会匹配到第一个 switch

分支。而这里没有其他特别的分支情况，所以 integerToDescribe 匹配到包含所有的 default 分支中。

当 switch 代码块执行完毕后，使用 println 函数打印该数字的描述。在这个例子中，数字 5 被准确地识别为一个质数。

6.6.4　return 语句

Return 语句：用在有反回值的方法中，用来返回方法指定类型的值，同时结束方法执行；当然如果没有返回值也可以用来终止方法运行；一般用于方法“func”里作为数据返回的情况比较多，使用情况可以参见后面章节的内容。

第 7 章
Swift 函数

本章将讲述 Swift 的函数，包含以下内容。

- 函数的定义和调用
- 函数的参数和返回值
- 函数的参数名称
- 函数类型
- 嵌套函数

函数是用来完成特定任务的独立的代码块。我们给一个函数起一个合适的名字，用来标识函数做什么，并且当函数需要执行时，这个名字会被“调用”。

Swift 统一的函数语法足够灵活，可以用来表示任何函数，包括从最简单的没有参数名字的 C 风格函数，到复杂的带局部和外部参数名的 Objective-C 风格函数。参数可以提供默认值，以简化函数调用。参数也可以既当作传入参数，也当作传出参数，也就是说，一旦函数执行结束，传入的参数值可以被修改。

在 Swift 中，每个函数都有一种类型，包括函数的参数值类型和返回值类型。我们可以把函数类型当作任何其他普通变量类型一样处理，这样就可以更简单地把函数当作别的函数的参数，也可以从其他函数中返回函数。函数的定义可以写在其他函数定义中，这样可以在嵌套函数范围内实现功能封装。

7.1 函数的定义和调用

当我们定义一个函数时，可以定义一个或多个有名字和类型的值，作为函数的输入（称参数，parameters），也可以定义某种类型的值作为函数执行结束的输出（称返回类型，return type）。

每个函数有个函数名，用来描述函数执行的任务。要使用一个函数时，我们用函数名“调用”，并传给它匹配的输入值（称实参，arguments）。一个函数的实参必须与函数参数表里参数的顺序一致。

下面例子中的函数称"sayHello"，之所以叫这个名字，是因为这个函数用一个人的名字当作输入，并返回给这个人的问候语。为了完成这个任务，我们定义一个输入参数，即一个叫作 personName 的 String 值，以及一个包含给这个人问候语的 String 类型的返回值：

```
func sayHello(personName:String)->String{
    let say = "你好，\(personName)！"
    return say
}
print(sayHello(personName: "雨燕！"))
```

输出：

你好，雨燕！

所有的这些信息汇总起来成为函数的定义，并以 func 作为前缀。指定函数返回类型时，用返回箭头 ->（一个连字符后跟一个右尖括号）后跟返回类型的名称的方式来表示。该定义描述了函数做什么，它期望接收什么和执行结束时它返回的结果是什么。这样的定义使得函数可以在别的地方以一种清晰的方式被调用。

调用“sayHello”函数时，在圆括号中传给它一个 String 类型的实参。因为这个函数返回一个 String 类型的值，“sayHello”可以被包含在 print 的调用中，用来输出这个函数的返回值。

7.2　函数的参数和返回值

函数参数和返回值在 Swift 中极为灵活。我们可以定义任何类型的函数，包括从只带一个未名参数的简单函数到复杂的带有表达性参数名和不同参数选项的复杂函数。

7.2.1　多个输入参数的函数

函数可以有多个输入参数，写在圆括号中，用逗号分隔，如：

```
func getMoney(工资:Int,扣税:Int)->Int{
      return 工资 - 扣税
}
print("波波每月实际收入：\(getMoney(工资: 30000,扣税: 5000))")
```

输出：

波波每月实际收入：25000

7.2.2　没有参数的函数

函数可以没有参数。下面这个函数就是一个无参函数，当它被调用时，返回固定的 String 消息：

```
func sayHi()->String{
    return "Hi,波波"
}
print(sayHi())
```

输出：

Hi,波波

7.2.3 没有返回值的函数

函数可以没有返回值。下面是 sayHello 函数的另一个版本，称 sayHi，这个函数直接输出 String 值，而不是返回它。

```
func HiMan(name:String){
    print("你好啊，\(name)")
}
HiMan(name: "老镇")
```

输出：

你好啊，老镇

7.2.4 没有返回值的函数调用具有返回值的函数

没有返回值的函数可以调用具有返回值的函数，如：

```
func printAndCount(stringToPrint:String)->Int{
     print(stringToPrint)
     return stringToPrint.count
}
func printWithoutCount(stringToPrint:String){
     printAndCount(stringToPrint: stringToPrint)
}
printWithoutCount(stringToPrint: "中澳通好")   //没有返回值，只打印了参数
```

输出：

中澳通好

第一个函数 “printAndCount”，输出一个字符串并返回 Int 类型的字符数。第二个函数“printWithoutCount”调用了第一个函数，但忽略了它的返回值。当第二个函数被调用时，消息依然会由第一个函数输出，但返回值不会被用到。

7.2.5 多个返回值的函数

我们可以用元组（tuple）类型让多个值作为一个复合值从函数中返回，如：

```
func getMinMax(array:[Int])->(min:Int,max:Int){
     var min = array[0]
     var max = array[0]
     //迭代数组，获取最小值和最大值
     for value in array{
          if value < min{
               min = value
          }
          if value > max{
               max = value
          }
```

```
        }

        return (min,max)
}
let values = getMinMax(array: [10,42,8,99,3])
print("最少值是：\(values.min)，最大值是：\(values.max)")
```

输出：

最少值是：3，最大值是：99

7.2.6　元组可选类型返回值的函数

返回多个值，我们使用元组，并且添加可选类型，如：

```
func getMinMaxMoney(array:[Int])->(min:Int,max:Int)?{
    if array.isEmpty {
          return nil
    }
    var min = array[0]
    var max = array[0]
    //迭代数组，获取最小值和最大值
    for value in array{
         if value < min{
               min = value
         }
         if value > max{
               max = value
         }
    }

    return (min,max)
}
var arrays = [Int]()
arrays = [8000,9000,7000]
//可选类型判断拆包
if let moneys = getMinMaxMoney(array: arrays){
     print("最少工资是：\(moneys.min)，最多工资是：\(moneys.max)")
}else{
     print("没有工资。")
}
```

输出：

最少工资是：7000，最多工资是：9000

7.3 函数的参数名称

函数的参数，一般有外部参数名称（我们称为参数别名）和本地参数名称（我们称为参数）。外部参数名称（参数别名）用于更好地让函数调用者理解这一参数的意思。本地参数名称（参数）用于在函数体中被实际调用。

7.3.1 外部参数名称

在调用函数时，给每个参数命名是非常有用的，因为这些参数名可以指出各个实参的用途是什么。

如果我们希望函数的使用者在调用函数时提供参数名字，那么就需要给每个参数除了局部参数名外再定义一个外部参数名。外部参数名写在局部参数名之前，用空格分隔，如：

```
    func join(string s1:String, toString s2:String, WithJoiner joiner:String)->
String{
         return s1+joiner+s2
    }
    print(join(string: "华仔", toString: "雨燕", WithJoiner: "讲"))
    输出：
    华仔讲雨燕
```

7.3.2 带默认值参数的函数

我们可以在函数体中为每个参数定义默认值。当默认值被定义后，为调用这个函数时，如果没有参数值，则使用参数定义的默认值，如：

```
    func isStudent(名字 userName:String, 年龄 age:Int = 10)->Bool{
        var 答案:Bool = false
        if(age < 30){
            答案 = true
        }
        if(userName == "张三" || userName == "李四" || userName == "王五"){
            答案 = true
        }
        print("\(userName) 是否学生：\(答案)")
        return 答案
    }
    isStudent(名字: "华仔")
    isStudent(名字: "张三", 年龄:41)
    输出：
    华仔 是否学生：true
    张三 是否学生：true
```

7.3.3　可变参数的函数

一个可变参数“variadic parameter”可以接受一个或多个值。当函数调用时，我们可以用可变参数来传入不确定数量的输入参数。通过在变量类型名后面加入“…”的方式来定义可变参数。

传入可变参数的值在函数体内当作这个类型的一个数组。例如，一个叫作 numbers 的 Int...型可变参数，在函数体内可以当作一个叫 numbers 的 Int[]型的数组常量，如下面的选秀例子。

```
func talentShow(numbers:Int...)->String{
     var total : Int = 0
     var min = numbers[0];
     var max = numbers[0];
     for number in numbers{
          total += number
          if(number > max){
               max = number
          }
          if(number < min){
               min = number
          }
     }

     return "一共有\(total)位佳丽，其中第一位\(min)号，最后一位是：\(max)号"

}
print(talentShow(numbers: 1,3,21,4,5,88))
print(talentShow(numbers: 31,23,14,99,6,28))
```

输出：

一共有 122 位佳丽，其中第一位 1 号，最后一位是：88 号
一共有 201 位佳丽，其中第一位 6 号，最后一位是：99 号

7.3.4　常量参数和变量参数

函数参数默认是常量。试图在函数体中更改参数值将会导致编译错误。这意味着我们不能错误地更改参数值。

但有时，如果函数中有传入参数的变量值副本将是很有用的。我们可以通过指定一个或多个参数为变量参数，从而避免自己在函数中定义新的变量。变量参数不是常量，可以在函数中把它当作新的可修改副本来使用。

通过在参数名前加关键字 var 来定义变量参数，如：

```
    //string 这里当作可以变化的返回值和参数使用
    func alignRight(string: String, totalLength: Int, pad: Character) -> String {
        let amountToPad = totalLength - string.count
        if amountToPad < 1 {
            return string
        }
        let padString = String(pad)
        var rtStr = string
        for _ in 1...amountToPad {
            rtStr = padString + rtStr
        }
        return rtStr
    }
    let originalString = "hello"
    let paddedString = alignRight(string: originalString, totalLength: 10, pad:
"-")
    print(paddedString)
    输出：
    -----hello
```

这个例子中定义了一个新的叫作“alignRight”的函数，用来右对齐输入的字符串到一个长的输出字符串中。左侧空余的地方用指定的填充字符填充。在这个例子中，字符串"hello"被转换成了"-----hello"。

“alignRight”函数将参数“string”定义为变量参数。这意味着“string”现在可以作为一个局部变量，用传入的字符串值初始化，并且可以在函数体中进行操作。

“alignRight”函数首先计算出多少个字符需要被添加到“string”的左边，以右对齐到总的字符串中。这个值存在局部常量“amountToPad”中。然后，这个函数将“amountToPad”个字符填充到“string”左边，并返回结果。它使用了“string”这个变量参数来进行所有字符串操作。

7.3.5 输入和输出参数

变量参数，如上所述，仅能在函数体内被更改。如果我们想要一个函数可以修改参数的值，并想要在这些修改在函数调用结束后仍然存在，那么就应该把这个参数定义为输入–输出参数“In-Out Parameters”。

定义一个输入–输出参数时，在参数定义前加“inout”关键字。一个输入–输出参数有传入函数的值，这个值被函数修改，然后被传出函数替换原来的值。

我们只能将变量作为输入–输出参数，不能传入常量或字面量（literal value），因为这些量是不能被修改的。当传入的参数作为输入–输出参数时，需要在参数前加“&”符号，表示这个值可以被函数修改。

注意，输入–输出参数不能有默认值，而且可变参数不能用“inout”标记。如果用“inouz”

标记一个参数，这个参数不能被 var 或 let 标记。

下面例子中的“swapTwoInts”函数，有两个分别为 a 和 b 的输入–输出参数：

```
//这个函数是将 a 和 b 的值互换了
func swapTwoInts( a: inout Int, b: inout Int) {
    let temp = a
    a = b
    b = temp
}
var a1 = 100
var b1 = 50
//调用输出参数的值时要加&符号到参数
swapTwoInts(a: &a1, b: &b1)
print("a1: \(a1),   b1: \(b1)")
输出：
a1：50,   b1：100
```

7.4　函数类型

每个函数都有特定的函数类型，由函数的参数类型和返回类型组成。

7.4.1　使用函数类型

在 Swift 中，使用函数类型就像使用其他类型一样，如：

```
func add(a:Int,b:Int) -> Int{
    return a + b
}
func multipy(a:Int,b:Int) -> Int{
    return a * b
}
//可以这样定义函数变量
var tmp1 = add
var tmp2 : (Int,Int)->Int = multipy
//函数变量的使用
print("2 加 5 是\(tmp1(2,5))")
print("2 乘 5 是\(tmp2(2,5))")
输出：
2 加 5 是 7
2 乘 5 是 10
```

7.4.2　使用函数类型作为参数

当把函数类型当作参数使用时，我们可以将函数的一部分实现交给函数的调用者，如：

```
func showMultipy(a:Int,b:Int,multipy:(Int,Int)->Int){
    let result = multipy(a,b)
    print("\(a)乘以\(b)等于\(result)")
}
//调用具有函数类型作为参数的函数
showMultipy(a:5, b: 6, multipy: multipy)
输出：
5 乘以 6 等于 30
```

可以看出，“multipy”是个闭包（其实，函数只是闭包的一种特殊形式），“showMultipy”函数的部分实现，都交给了“multipy”这个闭包，闭包的具体用法会在第 8 章具体讲解。

7.4.3 使用函数类型作为返回值

我们可以用函数类型作为另一个函数的返回类型。我们需要做的是在返回箭头（->）后写一个完整的函数类型，如：

```
func addOne(number:Int)->Int{
     return number + 1
}
func minusOne(number:Int)->Int{
     return number - 1
}

func IsAddOrMinus(判断正负:Bool)->(Int)->Int{
     return 判断正负 ? addOne : minusOne
}
print("调用函数类型作为返回值：\(IsAddOrMinus(判断正负: 1>0)(10))")
输出：
调用函数类型作为返回值：11
```

可以看出，函数“判断加减”的参数“（1>0）”作为“IsAddOrMinus”使用的参数，然后后面的“（10）”，作为判断后调用“加一”或“减一”函数的参数。

7.4.4 使用函数类型作为常量

继续将上面例子的扩展，我们可以把函数类型作为常量使用，如：

```
var num = 10
let fuc = IsAddOrMinus(判断正负: num > 11)
while num != 0{
     print("\(num) ", terminator: " ")
     //调用常量函数
     num=fuc(num)
}
print("")输出：
10,9,8,7,6,5,4,3,2,1,
```

7.5　嵌套函数

本章中所见到的所有函数都称作全局函数（global functions），它们定义在全局域中。我们也可以把函数定义在别的函数体中，称作嵌套函数（nested functions）。

在默认情况下，嵌套函数是对外界不可见的，但可以被其封闭函数（enclosing function）来调用。一个封闭函数也可以返回它的某一个嵌套函数，使得这个函数可以在其他域中被使用，如：

```
func getResult(加薪减薪:Bool)->(Int,Int)->Int{
     func raises(工资:Int,加薪:Int)->Int{
          return 工资+加薪
     }
     func cut(工资:Int,减薪:Int)->Int{
          return 工资-减薪
     }

     return 加薪减薪 ? raises : cut
}
//定义常量函数
let fun = getResult(加薪减薪: true)
//调用嵌套的函数类型
var result = fun(9500,2000)
print("今年的工资是：\(result)")
输出：
今年的工资是：11500
```

第 8 章
Swift 闭包

本章将讲述 Swift 的闭包，包含以下内容。

- 闭包的定义
- 闭包表达式
- 挂尾闭包
- 闭包的值捕获
- 闭包是引用类型

8.1 闭包的定义

闭包是自包含的函数代码块，可以在代码中被传递和使用。Swift 中的闭包与 C 和 Objective-C 中的代码块及其他一些编程语言（C#）中的 lambdas 函数相似。

闭包能捕获和存储其所在上下文中任意常量和变量的引用。这就是所谓的闭合并包裹着这些常量和变量，俗称闭包。Swift 会为我们管理在捕获过程中涉及的所有内存操作（很智能）。

在第 7 章中介绍的全局函数和嵌套函数实际上也是特殊的闭包，闭包采取如下三种形式之一。

- 全局函数：一个有名字但不会捕获任何值的闭包。
- 嵌套函数：一个有名字并可以捕获其封闭函数域内值的闭包。
- 闭包表达式：一个利用轻量级语法所写的可以捕获其上下文中变量或常量值的匿名闭包。

闭包表达式拥有简洁的风格，Swift 鼓励开发者在常见场景中进行语法优化，主要优化如下。

- 利用上下文推断参数和返回值类型。
- 隐式返回单表达式闭包，即单表达式闭包可以省略 return 关键字。
- 参数名称缩写。
- 挂尾闭包的使用。

8.2　闭包表达式

闭包表达式是一种利用简洁语法构建内联闭包的方式。

闭包表达式提供了一些语法优化，使得编写闭包变得简单、明了。

下面讲述闭包表达式的例子，通过使用几次代码修改，展示 sorted 函数定义和语法优化的方式。每次修改，都进一步简化了代码并保留了相同的功能。（这是一种语法和编码方式的进步，使用更简洁的代码做更多的事情！）

8.2.1　sorted 函数

Swift 标准库提供了 sorted 函数，会根据我们提供的基于输出类型排序的闭包函数将已知类型数组中的值进行排序。一旦排序完成，函数会返回一个与原数组大小相同的新数组，该数组中包含已经正确排序的同类型元素。

下面的闭包表达式示例使用 sorted 函数对一个 String 类型的数组进行字母逆序排序，以下是初始数组值。

```
let city = ["Zhuhai","Guangzhou","Shenzhen","Dongguan","Fushan","Zhongshan"]
```

sorted 函数需要传入以下两个参数。

（1）已知类型的数组。

（2）闭包函数。该闭包函数需要传入与数组类型相同的两个值，并返回一个布尔类型值来告诉 sorted 函数当排序结束后传入的第一个参数排在第二个参数前面还是后面。如果第一个参数值出现在第二个参数值前面，排序闭包函数需要返回 true，反之返回 false。

本例对一个 String 类型的数组进行排序，因此排序闭包函数类型为（String-String）-> Bool。

提供排序闭包函数的一种方式是编写一个符合其类型要求的普通函数，并将其作为 sorted 函数的第二个参数传入：

```
func backward(s1:String, s2:String)->Bool{
     return s1 > s2;
}
var newCity1 = city.sorted(by: backward)
print("newCity1: \(newCity1)")
输出：
newCity1：[Zhuhai, Zhongshan, Shenzhen, Guangzhou, Fushan, Dongguan]
```

如果第一个字符串“s1”大于第二个字符串“s2”，backwards 函数返回 true，表示在新的数组中“s1”应该出现在“s2”前。

对于字符串中的字符来说，“大于”表示“按照字母顺序较晚出现”。这意味着字母“B”大于字母“A”，字符串“Tom”大于字符串“Tim”。其将进行字母逆序排序，“Zhongshan”将会排在“Shenzhen”之前。

然而，这是一个相当冗长的方式，本质上只是写了一个单表达式函数（a > b）。我们利用闭包表达式语法可以更好地构造一个内联排序闭包。

8.2.2 使用闭包表达式

闭包表达式语法的一般形式如下：

```
没有函数名称 {
                （参数）->返回类型 in
                代码段
            }
```

我们套用这一语法，编写代码如下。

```
var newCity2 = city.sorted(by: {
     (s1:String, s2:String) ->Bool in
     return s1 > s2
})
print("newCity2: \(newCity2)")
输出：
newCity2：[Zhuhai, Zhongshan, Shenzhen, Guangzhou, Fushan, Dongguan]
```

很明显，这里我们就使用了闭包表达式，在“sort”方法里面，直接通过闭包表达式把 8.2.1 节代码中定义的“func backward”这个函数定义删除了，简介了对一个函数的定义。闭包的函数体部分由关键字 in 引入，该关键字表示闭包的参数和返回值类型定义已经完成，闭包函数体即将开始。

8.2.3 根据上下文推断参数

因为排序闭包函数是作为 sorted 函数的参数传入的，所以 Swift 可以推断其参数和返回值的类型。sorted 期望第二个参数是类型为（String-String）-> Bool 的函数，因此实际上 String-String 和 Bool 类型并不需要作为闭包表达式定义中的一部分。 因为所有的类型都可以被正确推断，返回箭头（->）和围绕在参数周围的括号也可以省略，如：

```
var newCity3 = city.sorted(by: {
     s1, s2 in return s1 > s2})
print("newCity3: \(newCity3)")
输出：
newCity3：[Zhuhai, Zhongshan, Shenzhen, Guangzhou, Fushan, Dongguan]
```

实际上，在任何情况下，通过内联闭包表达式构造的闭包作为参数传递给函数时，都可以推断出闭包的参数和返回值类型，这意味着，我们几乎不需要利用完整格式构造任何内联闭包。

8.2.4 单表达式闭包省掉返回关键字

如果闭包的语句只有单行表达式，可以把 return 关键字省略，如：

```
var newCity4 = city.sorted(by: {
     s1, s2 in s1 > s2})
print("newCity4: \(newCity4)")
输出：
newCity4：[Zhuhai, Zhongshan, Shenzhen, Guangzhou, Fushan, Dongguan]
```

在这个例子中，sorted 函数的第二个参数函数类型明确了闭包必须返回一个 Bool 类型值。因为闭包函数体只包含了一个单一表达式“s1 > s2”，该表达式返回 Bool 类型值，因此这里没有歧义，return 关键字可以省略。

8.2.5　参数名称缩写

Swift 自动为内联函数提供了参数名称缩写功能，我们可以直接通过“$0，$1，$2”（0 是第一个参数）来顺序调用闭包的参数。

如果在闭包表达式中使用参数名称缩写，我们可以在闭包参数列表中省略对其的定义，并且对应参数名称缩写的类型会通过函数类型进行推断。in 关键字也同样可以省略，因为此时闭包表达式完全由闭包函数体构成，如：

```
var newCity5 = city.sorted(by: {
     $0 > $1})
print("newCity5: \(newCity5)")
输出：
newCity5：[Zhuhai, Zhongshan, Shenzhen, Guangzhou, Fushan, Dongguan]
```

8.2.6　运算符函数

实际上还有一种更简短的方式来编写上述例子中的闭包表达式。Swift 的 String 类型定义了关于大于号“>”的字符串实现，其作为一个函数接受两个 String 类型的参数并返回 Bool 类型的值。而这正好与 sorted 函数的第二个参数需要的函数类型相符合。因此，我们可以简单地传递一个大于号，Swift 可以自动推断出我们想使用大于号的字符串函数实现，如：

```
var newCity6 = city.sorted(by: >)
print("newCity6: \(newCity6)")
输出：
newCity6：[Zhuhai, Zhongshan, Shenzhen, Guangzhou, Fushan, Dongguan]
```

8.3　挂尾闭包

如果闭包是函数的最后一个参数，则可以把闭包移到函数后面，这种方法用于闭包表达式语句比较多的情况，如：

```
let numbers = [1800,756,9800]

//挂尾闭包
```

```
let strResult = numbers.map{
      (number: Int) -> String in
      var outPut : String = ""
      var num = number

      while(num > 0){
           outPut = ditNumber[num % 10]! + outPut
           num /= 10
      }

      return outPut
}
print("转换为汉字的数组是：\(strResult)")
输出：
转换为汉字的数组是：[一八零零，七五六，九八零零]
```

上例中尾随闭包语法在函数后整洁封装了具体的闭包功能，而不再需要将整个闭包包裹在 map 函数的括号内。

8.4 闭包的值捕获

闭包可以在其定义的上下文中捕获常量或变量。即使定义这些常量和变量的原域已经不存在，闭包仍然可以在闭包函数体内引用和修改这些值。

Swift 最简单的闭包形式是嵌套函数，也就是定义在其他函数体内的函数。嵌套函数可以捕获其外部函数所有的参数及定义的常量和变量，如：

```
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
     var runningTotal = 0
     func incrementer() -> Int {
          runningTotal += amount
          return runningTotal
     }
     return incrementer
}

let incrementByTen = makeIncrementer(forIncrement: 10)
print(incrementByTen())
print(incrementByTen())
print(incrementByTen())
//因为闭包是一个引用类型
输出：
```

```
10
20
30
```

8.5　闭包是引用类型

在上面的例子中，incrementByTen 是常量，但常量指向的闭包仍然可以增加其捕获的变量值。这是因为函数和闭包都是引用类型。

无论我们将函数或闭包赋值给一个常量还是变量，实际上都是将常量或变量的值设置为对应函数或闭包的引用。在上面的例子中，incrementByTen 指向闭包的引用是一个常量，而并非闭包内容本身。这也意味着，如果我们将闭包赋值给了两个不同的常量/变量，两个值都会指向同一个闭包。

第 9 章
Swift 枚举

本章将讲述 Swift 的枚举，包含以下内容。

- 枚举的定义、结构和使用
- 枚举在 swich 语句中的使用
- 枚举关联值
- 枚举原始值

如果读者熟悉 C 语言，就会知道，C 语言中枚举指定相关名称为一组整型值。Swift 中的枚举更加灵活，不必给每一个枚举成员提供一个值。如果一个值（被认为是“原始”值）被提供给每个枚举成员，则该值可以是一个字符串、一个字符、一个整型值或浮点值。很明显，枚举属于值类型。

9.1 枚举的定义、结构和使用

枚举定义了一个通用类型的一组相关的值，使我们可以在自己的代码中以一个安全的方式来使用这些值。

枚举的结构是：

```
enum SomeEnumeration {
    // enumeration definition goes here
}
```

枚举一般这样使用：

```
//定义枚举，里面的 case 看上去就是为 switch 准备的，天生一对
enum direction{
     case 东
     case 南
     case 西
     case 北
}
//调用枚举
var direct = direction.东
print("现在的方向是：\(direct)")
```

输出：

现在的方向是：方向 . 东

从上面的代码可以看出，枚举值是通过 case 关键字来定义的，可以看到枚举和 switch 简直就是天生一对！

9.2　枚举在 switch 语句中的使用

接着使用上面的枚举定义，我们添加代码如下：

```
//在 switch 中使用
switch direct {
case direction.东: print("问君能有几多愁，恰似一江春水向东流。");
case direction.南: print("红豆生南国，春来发几枝？ ")
case .西:  print("故人西辞黄鹤楼，烟花三月下扬州。")
case .北:  print("北国风光，千里冰封，万里雪飘。")
}
```

输出：

问君能有几多愁，恰似一江春水向东流。

9.3　枚举关联值

我们可以定义 Swift 的枚举存储任何类型的相关值，如果需要，每个成员的数据类型可以是各不相同的。枚举的这种特性与其他语言中的可辨识联合（discriminated unions）、标签联合（tagged unions）或变体（variants）相似。

例如，假设一个库存跟踪系统需要利用两种不同类型的条形码来跟踪商品。有些商品上标有 UPC-A 格式的一维码，它使用数字 0～9。每个条形码都有一个代表“数字系统”的数字，该数字后接 10 个代表“标识符”的数字。最后一个数字是“检查”位，用来验证代码是否被正确扫描，如图 9-1 所示。

其他商品上标有 QR 码格式的二维码，它可以使用任何 ISO 8859-1 字符，并可以编码一个最多拥有 2953 字符的字符串，如图 9-2 所示。



图 9-1



图 9-2

对于库存跟踪系统来说，能够把 UPC-A 码作为三个整型值的元组，以及把 QR 码作为一个任意长度的字符串存储起来都是方便的，如：

```
enum Barcode{
    case UPCA(Int,Int,Int,Int)
    case QRCode(String)
}

var goods = Barcode.UPCA(8, 85909, 51226, 3)
goods = Barcode.QRCode("麦咖啡套餐")

switch goods{
case let .UPCA(系统位,厂商位,商品位,校验位):
    print("一维码：\(系统位)\(厂商位)\(商品位)\(校验位)")
case let .QRCode(产品编码):
    print("二维码：\(产品编码)")
}
输出：
二维码：麦咖啡套餐
```

9.4 枚举原始值

作为相关值的替代物，枚举成员可以被默认值（称原始值）预先填充，这些原始值具有相同的类型，如：

```
enum planet: Int{
    case 水星 = 1
    case 金星
    case 地球
    case 火星
    case 木星
    case 土星
    case 天王星
    case 海王星
    case 冥王星
}
//返回个数（从 0 开始）
print(planet.冥王星.rawValue)
let onePlanet = planet.天王星
print(onePlanet.hashValue)
输出：
9
6
```

第 10 章
Swift 类和结构体

本章将讲述 Swift 的类和结构体，包含以下内容。

- 类和结构体的对比
- 结构体是值类型
- 类是引用类型
- 如何选择类和结构体
- 集合类型的赋值和复制

类和结构体是人们构建代码所用的一种通用且灵活的构造体。为了在类和结构体中实现各种功能，我们必须严格按照常量、变量及函数所规定的语法规则来定义属性和添加方法。

与其他编程语言不同的是，Swift 并不要求我们为自定义的类和结构去创建独立的接口和实现文件。我们所要做的只是在一个单一文件中定义一个类或结构体，系统将会自动生成面向其他代码的外部接口。

10.1 类和结构体的对比

在 Swift 中，类和结构体有以下共同点。

- 定义属性：用于存储值。
- 定义方法：用于提供功能。
- 定义下标：用于访问值。
- 定义构造函数：用于初始化相关属性状态。
- 通过扩展：增加默认实现的功能。
- 通过协议：提供某种功能标准。

类与结构体相比，还有以下功能。

- 继承：使一个类能继续另一个类的特性。
- 类型检查：可以在实例运行时检查类型和对其进行解析。
- 析构：可以释放实例所占用的资源。
- 引用计数器：使类可以被多次引用。

10.1.1 类和结构体的定义

类和结构体有着相似的定义方式。我们可以通过关键字“class”和“struct”来分别表示类和结构体，并在一对花括号中定义它们的具体内容，如：

```
struct weapon{                  //武器
    var name = ""               //武器名称
    var hitRate = 0.0           //命中率
    var attack = 0              //攻击力
}

class Player {                  //游戏角色
    var name = ""               //姓名
    var hp = 0                  //血量
    var profession = ""         //职业
    var sabre = weapon()        //佩刀
}
```

在上述代码中，我们定义了“武器”这个结构体，属性分别有“武器名称”、“命中率”、“攻击力”；然后，我们定义了一个“游戏角色”类，里面的属性分别有“姓名”、“血量”、“职业”、“佩刀”，“佩刀”定义为结构体“武器”的实例。

10.1.2 类和结构体的实例

我们可以这样创建上面定义好的结构体和类的实例，如：

```
var 青龙刀 = weapon() //结构体都是值类型
let 关小羽 = Player() //类都是引用类型
```

结构体和类都使用构造器语法来生成新的实例。构造器语法的最简单形式是在结构体或类的类型名称后跟随一个空的圆括号，如“weapon()”或“Player()”。通过这种方式所创建的类或结构体实例，其属性均会被初始化为默认值。示意图如图 10-1 所示。



图 10-1

10.1.3 类和结构体的属性访问

通过使用点号“.”，我们可以访问实例中所含有的属性。其语法规则是，实例名后面紧跟属性名，两者通过点号“.”连接，如：

```
青龙刀.name = "青龙刀"
青龙刀.hitRate = 0.9
青龙刀.attack = 800

关小羽.name = "关小羽"
关小羽.hp = 8000
关小羽.profession = "战士"
关小羽.sabre = 青龙刀
```

10.2　结构体是值类型

值类型：被赋给一个变量、常数或本身被传递给一个函数时，实际上操作的是它的复本（简单来说，值类型传递的是值复本）。

在之前的章节中，我们已经大量使用了值类型。实际上，在 Swift 中，所有的基本类型——整数、浮点数、布尔值、字符串、数组、集、字典和枚举，都是值类型，并都是以结构体的形式来实现的。

在 Swift 中，所有的结构体都是值类型。这意味着它们的实例，以及实例中所包含的任何值类型属性，在代码中传递时都会被复制，如：

```
关小羽.sabre.attack = 10000
print("青龙刀杀敌扣血:\(青龙刀.attack)")
print("关小羽佩刀杀敌扣血:\(关小羽.sabre.attack)")
输出：
青龙刀杀敌扣血:800
关小羽佩刀杀敌扣血:10000
```

上例中“关小羽的 sabre 的 attack 属性”这个结构体赋予新的值“10000”，但是，实际上它被赋予的是值的一个复本。赋值过程结束后并不影响结构体“青龙刀的 attack”所储存的原始值的复本，所以，结构体“青龙刀”原来“attack”属性的值还是“800”，而修改后的“关小羽”所包含的结构体“sabre”的“attack”属性是“10000”。这就说明了结构体是值类型，每次创建实例都是值的复本，不影响原来结构体的属性。

10.3　类是引用类型

引用类型：在被赋予一个变量、常量或被传递到一个函数时，操作的是引用，其并不是复制。因此，引用的是已存在的实例本身而不是其复本，如：

```
let 冰封关羽 = 关小羽
冰封关羽.hp = 1000
print("新皮肤冰封关羽血量:\(冰封关羽.hp)")
        print("关小羽的血量:\(关小羽.hp)")
```

上例中，“关小羽”和常量“冰封关羽”明显是同一个实例的不同引用，“冰封关羽”的属性“血量”增加到“1000”时，也就是“关小羽”的属性“血量”增加到“1000”。

10.4　如何选择类和结构体

我们可以看到，类和结构体有很多相似的地方，总的来说，以下这些情况，可以考虑使用结构体。

- 封装一些简单的数据值。
- 希望在赋值或传递时，使用值复本而不是值传递。
- 希望实例使用值类型，而非引用类型。
- 不希望继承我们创建的结构体（结构体不能被继承）。

一般情况下，我们还是使用类的情况比较多。

10.5 集合类型的赋值和复制

Swift 中字符串、数组、集和字典类型均以结构体的形式实现。这意味着 String、Set、Array、Dictionary 类型数据被赋值给新的常量（或变量），或者当被传入函数（或方法）中时，它们的值会发生复制行为（值传递方式）。

第 11 章
Swift 属性

本章将讲述 Swift 的属性，包含以下内容。

- 存储属性
- 计算属性
- 属性观察器
- 全局变量和局部变量
- 静态属性

属性将值与特定的类、结构或枚举关联。存储属性：存储常量或变量作为实例的一部分；计算属性：计算（而不是存储）一个值。计算属性可以用于类、结构体和枚举中，存储属性只能用于类和结构体。

存储属性和计算属性通常用于特定类型的实例，但是，属性也可以直接用于类型本身，这种属性称为类型属性。

另外，还可以定义属性观察器来监控属性值的变化（这点真是太人性化了），以此来触发一个自定义的操作。属性观察器可以添加到自己写的存储属性上，也可以添加到从父类继承的属性上。

11.1 存储属性

简单来说，一个存储属性就是存储在特定类或结构体的实例里的一个常量或变量，存储属性可以是“变量存储属性”（用关键字 var 定义），也可以是“常量存储属性”（用关键字 let 定义）。

11.1.1 普通存储属性

可以在定义存储属性时指定默认值，甚至修改变量存储属性的值（常量一旦赋值就不能修改），如：

```
class player {
    let NO: String = "24"
    var score: Int = 60
}
```

```
let kobe = player()
print("Kobe 的球衣号码是：\(kobe.NO)，最后一场比赛得分是：\(kobe.score)")
kobe.score = 81
print("职业生涯最高得分是：\(kobe.score)")
输出：
Kobe 的球衣号码是：24，最后一场比赛得分是：60
职业生涯最高得分是：81
```

上述例子定义了一个“player”类，里面有字符串常量属性“NO”，默认值为“24”，还有一个整型变量属性“score”，默认值是“60”。创建“player”的对象实例“kobe”后，可以调用实例“kobe”的属性，输出相关值，如“kobe.score”可以看到输出的“score”属性值是“60”。由于“score”属性是变量存储属性，所以可以进行修改，我们修改“score”存储属性为“81”，输出为“81”。但是，我们并不能修改常量属性“NO”，因为常量一旦赋值就不能修改了，若强制修改，编译器会报错。

11.1.2 延迟存储属性

延迟存储属性是指当第一次被调用时才会计算其初始值的属性。在属性声明前使用lazy 来标识一个延迟存储属性，如：

```
class map{
    var place = "死亡沙漠"
}

class role{
    lazy var battleMap = map()      //延迟创建对象
    var online = true
}

let master = role()
print("法师到了 \(master.battleMap.place)")     //使用时才创建实际对象
输出：
法师到了 死亡沙漠.map
```

从上述例子中可以看到，当创建“master”实例后，调用“battleMap”属性，才开始创建“battleMap”里的“map”实例。这样做的好处是避免类的不必要的初始化，减少资源的浪费。

11.2 计算属性

除存储属性外，类、结构体和枚举可以定义“计算属性”，计算属性不直接存储值，而是提供一个“getter”来获取值和一个可选的“setter”来设置值，如：

```
struct 坐标{
     var x = 0, y = 0
}
struct 步伐 {
     var walk = 0, run = 0
}

struct 人物斜上移动 {
     var 人物坐标 = 坐标()
     var 人物步伐 = 步伐()

     var 走坐标: 坐标{
          get{
              let newX = 人物坐标.x + 人物步伐.walk
              let newY = 人物坐标.y + 人物步伐.walk

              return 坐标(x: newX, y: newY)
          }
          set{
              人物坐标.x = newValue.x - 人物步伐.walk
              人物坐标.y = newValue.y - 人物步伐.walk
          }
     }
}
var 某次移动 = 人物斜上移动(人物坐标: 坐标(), 人物步伐: 步伐(walk: 10, run: 20))
print("第一次移动后的坐标：x: \(某次移动.走坐标.x), y:\(某次移动.走坐标.y)")
print("第一次移动后的人物坐标：x: \(某次移动.人物坐标.x), y:\(某次移动.人物坐
标.y)")

某次移动.走坐标 = 坐标(x: 30, y: 30)
print("再一次移动后的坐标：x: \(某次移动.走坐标.x), y:\(某次移动.走坐标.y)")
print("再一次移动后的人物坐标：x: \(某次移动.人物坐标.x), y:\(某次移动.人物坐
标.y)")
```

输出：

第一次移动后的坐标：x: 10，y:10
第一次移动后的人物坐标：x: 0，y:0
再一次移动后的坐标：x: 30，y:30
再一次移动后的人物坐标：x: 20，y:20

从上述例子可以看到，“get”用来返回“坐标”结构体，“set”用来设置“坐标”结构体的“x”和“y”的值，这里的“set”没有定义新的参数名，所以我们可以直接使用默认名称“newValue”。

当然，还有一种情况，就是不能修改属性的值，只能获取属性的值，在这种情况下，我们称之为“只读属性”，如：

```
struct 战士 {
      var 武器攻击力 = 0, 头盔攻击力 = 0, 戒指攻击力 = 0, 盔甲攻击力 = 0
      var 经验倍数 = 1
      var 总攻击力 : Int{
           return 经验倍数 * (武器攻击力 + 头盔攻击力 + 戒指攻击力 + 盔甲攻击力)
      }
}
let 发思特战士 = 战士(武器攻击力: 30, 头盔攻击力: 2, 戒指攻击力: 4, 盔甲攻击力: 10,
经验倍数: 3)
print("发思特战士的总攻击力: \(发思特战士.总攻击力)")
输出
发思特战士的总攻击力：138
```

从上述例子可以看到，“总攻击力”是一个只读属性，只有“return”返回值的属性，如果强行给“发思特战士.总攻击力”属性赋值，编译器将会报错。

11.3 属性观察器

属性观察器：监控和响应属性值的变化，每次属性被更改值时，都会调用属性观察器，甚至新的值和现在的值相同时也一样会调用（简单来说，就是一直观察我们定义的属性，就算不断赋予同一个值，也会一直观察）。

可以为属性添加如下的一个或全部观察器。

- willSet：在设置新的值之前调用（新值更改前调用）。
- didSet：在新的值被设置之后立即调用（新值更改后调用）。

“willSet”观察器会将新的属性值作为固定参数传入，在“willSet”的实现代码中可以为这个参数指定一个名称，如果不指定则使用默认名称“newValue”。

“didSet”类似“willSet”观察器，会将旧的属性值作为参数传入，可以为该参数命名或使用默认参数名“oldValue”，如：

```
class 经验计算{
     var 总经验 : Int = 0{
          willSet{
               print("当前经验是: \(newValue), ",terminator:"")
          }
          didSet{
               print("增加经验\(总经验 - oldValue)")
          }
     }
}
var MU游戏经验计算 = 经验计算()
MU游戏经验计算.总经验 = 1000
MU游戏经验计算.总经验 = 1200
```

```
MU 游戏经验计算.总经验 = 1200
输出：
当前经验是：1000，增加经验 1000
当前经验是：1200，增加经验 200
当前经验是：1200，增加经验 0
```

在上述例子中，“总经验”这个属性默认值是“0”，使用了“属性观察器”，当第一次被赋予新的值“1000”时，触发了“willSet”，输出了“当前经验是：1000”，在被赋值后，执行“didSet”，输出了“增加经验 1000”。第二次被赋予新的值“1200”时，也同样执行第一次的流程。第三次，我们即使赋予相同的值“1200”，也是会执行和前两次一样的流程，执行“didSet”方法，计算出“增加经验 0”。

11.4　全局变量和局部变量

计算属性和属性观察器所描述的模式也可用于全局变量和局部变量。

- 全局变量：在函数、方法、闭包或任何类型外定义的变量。
- 局部变量：在函数、方法、闭包内部定义的变量。

前面章节提到的全局变量或局部变量都属于存储型变量，与存储属性类似，它提供特定类型的存储空间，并允许读取和写入。另外，在全局或局部范围都可以定义计算型变量，以及为存储型变量定义观察器，计算型变量与计算属性一样，返回一个计算的值而不是存储值，声明格式也完全一样。

11.5　静态属性

静态属性：使用关键字 static 来定义值类型的类型属性，如：

```
struct 背包{
      static var 最大负载 = 300
      var 牌子 : String = "优秀衣库"
}
背包.最大负载 = 600
print("华仔的背包的最大负载是：\(背包.最大负载)" )
输出：
华仔的背包的最大负载是：600
```

在上述例子中，“最大负载”属性前添加了“static”关键字，以说明是“静态属性”，与实例的属性一样，静态属性的访问也是通过点“.”运算符来进行的，但是，静态属性是通过类型本身来获取和设置的，而不是通过实例。例如，我们可以直接通过结构体“背包.最大负载”来设置静态属性的值。但是，非静态属性“牌子”就不能通过“背包.牌子”直接设置属性了，如果强行设置，编译器会报错。

第 12 章
Swift 方法

本章将讲述 Swift 的方法，包含以下内容。

- 实例方法
- 静态方法

方法是与某些特定类型相关联的函数。类、结构体、枚举都可以定义实例方法，也可以定义静态方法。

12.1 实例方法

实例方法：给特定类型实例封装的具体功能函数。

实例方法适用于类、结构体、枚举。

12.1.1 使用类中定义的实例方法

首先在类中定义方法，然后通过创建类的实例，使用“实例.方法名”即可调用，如：

```
    enum 怪物经验对应列表:Int{
        case 骷髅兵 = 10, 毒虫 = 20, 独眼巨人 = 30, 幽灵 = 40, 骷髅弓箭手 = 50, 地狱猎犬怪 = 60, 地狱蜘蛛 = 70, 骷髅战士 = 80, 死灵法师 = 90, 毒牛怪 = 100, 暗黑骑士 = 200, 魔鬼戈登 = 500
    }

    struct 服务器经验倍数 {
         var 开启 = false
         var 倍数 = 0
    }

    class 人民币玩家{
         var 经验值 = 0
         var 服倍数 = 服务器经验倍数()

         func 挂机经验(){
```

```
            经验值 += 200
            print("当前角色的挂机经验值是：\(经验值)")
        }

        func 打怪经验(怪物经验值: Int, 经验倍数: Int){
            经验值 += (怪物经验值 * 经验倍数)
            if(服倍数.开启 && 服倍数.倍数 > 1){
                经验值 *= self.服倍数.倍数
            }
            print("当前角色的打怪经验值是：\(经验值)")
        }

    }


    var 人民币法师 = 人民币玩家()
    人民币法师.挂机经验()
    人民币法师.打怪经验(怪物经验值: 怪物经验对应列表.暗黑骑士.rawValue, 经验倍数: 3)
    人民币法师.服倍数.开启 = true
    人民币法师.服倍数.倍数 = 2
    人民币法师.打怪经验(怪物经验值: 怪物经验对应列表.魔鬼戈登.rawValue, 经验倍数: 3)
    输出：
    当前角色的挂机经验值是：200
    当前角色的打怪经验值是：800
    当前角色的打怪经验值是：4600
```

在上述例子中，我们定义了一个“怪物经验对应列表”的枚举、一个“服务器经验倍数”的结构体和一个“人民币玩家”的类。类里面定义了两个属性，分别是“经验值”和结构体“服务器经验倍数”的实例属性“服倍数”，定义了两个实例方法，分别是没有参数的“挂机经验”和具有别名参数的“打怪经验”。里面分别计算出挂机和打怪的经验计算。

当创建“人民币法师”这个实例后，我们先开始挂机，通过“人民币法师.挂机经验”调用了“挂机经验”这个无参数的实例函数，获取了“200”经验值。

挂机后，我们开始打怪，刚开始，先不挑战 BOSS，先试下打比较厉害的小怪看看能否应付。我们选择了“暗黑骑士”，通过调用实例方法“打怪经验”，参数分别为“暗黑骑士”和“3 倍经验”，挑战成功，我们获取了“800”经验值。

最后，我们要挑战 BOSS “魔鬼戈登”，实例属性“服倍数”开启，打怪的经验倍数增加到“2 倍”，再调用实例方法“打怪经验”，获取了“4600”经验值。

12.1.2　使用结构体中定义的实例方法

首先在结构体中定义方法，但定义方法前要添加“mutating”关键字来声明这个方法是

“变异方法”，如果在此方法中还要调用结构体中的属性，则必须在调用属性前添加“self.”来证明属性来源于此结构体。然后通过创建类的实例，使用“实例.方法名”即可调用，如：

```
struct 角色坐标 {
    var x = 0, y = 0

    mutating func 安全区坐标(横坐标:Int,纵坐标:Int){
        self.x = 横坐标
        self.y = 纵坐标
        print("你所移动到的安全区坐标是：x: \(self.x), y: \(self.y)")
    }

}
var 某坐标 = 角色坐标()
某坐标.安全区坐标(横坐标: 500, 纵坐标: 500)
输出：
你所移动到的安全区坐标是：x: 500, y: 500
```

在上述例子中，必须给结构体里的“安全区坐标”这个实例方法添加“mutating”关键字，否则，通过结构体实例来调用此方法编译器会报错；在此方法里调用结构体的“x”和“y”属性也必须添加“self.”来调用，否则编译器也会报错。

12.1.3 使用枚举中定义的实例方法

枚举中定义的实例方法也和结构体一样，要在定义的方法前添加“mutating”关键字来声明这个方法是“变异方法”，如果在此方法中还要调用枚举中的匹配枚举值，则必须使用“self”来证明匹配枚举值来源于此枚举。然后通过创建类的实例，使用“实例.方法名”即可调用，如：

```
enum 猜拳{
    case 剪刀,石头,布
    mutating func 出拳(){
        //print("我出拳：\(self), ",false)
        print("我出拳：\(self), ",terminator:"")
        switch self{
        case .剪刀: self = .石头
        case .石头: self = .布
        case .布: self = .剪刀
        }
        print("你要赢我，你出拳：\(self)")
    }
}

var 我出拳 = 猜拳.剪刀
```

```
我出拳.出拳()
输出：
出拳：剪刀，你要赢我，你出拳：石头
```

在上述例子中，我们定义了一个枚举“猜拳”，匹配的枚举值是“剪刀”、“石头”、“布”，然后再定义一个枚举的实例方法“出拳”，必须添加关键字“mutating”，否则编译器会报错。此方法中先是打印了我出拳的匹配枚举，然后对应你要赢我出的拳，必须做出的对应匹配枚举值，例如，我出“剪刀”你出“石头”，我出“石头”你出“布”，我出“布”你出“剪刀”，这样你才会赢。

然后通过创建枚举实例“我出拳”为“猜拳.剪刀”，调用实例的“我出拳.出拳”方法，即可获取相关结果了。

12.2　静态方法

静态方法：不用创建特定类型的实例就能调用的具体功能函数。

静态方法适用于类、结构体。

12.2.1　使用类中定义的静态方法

首先在类中定义方法，通过在方法名前添加关键字“class”（本版本中添加 static 也可以），然后使用“类名.方法名”即可调用，如果方法中包含调用属性，那么此属性必须是静态属性（静态属性也是在普通属性前添加关键字 static），如：

```
class 玩家{
    static var 姓名 = ""
    class func 服务器名(){
        print("\(姓名) 在珠海电信一区")
    }

}
玩家.姓名 = "上官卫生"
玩家.服务器名()
输出：
上官卫生 在珠海电信一区
```

12.2.2　使用结构体中定义的静态方法

和在类中使用的静态方法类似，通过在方法名前添加关键字“static”，然后使用“结构体.方法名”即可调用，如果方法中包含调用属性，那么此属性必须是静态属性（静态属性也是在普通属性前添加关键字 static），如:

```
struct 复活{
```

```
        static var HP = 0
        static func 复活点(当前区域:String){
            print("\(当前区域 )安全区，你当前 HP 为：\(HP)")
        }
}
复活.复活点(当前区域： "死亡沙漠")
输出：
死亡沙漠安全区，你当前 HP 为：0
```

第 13 章
Swift 下标

本章将讲述 Swift 的下标，包含以下内容。

- 下标的定义
- 下标的用法
- 下标脚本

13.1 下标的定义

下标：类、结构、枚举这些集合、列表或序列的成员列表的快捷访问方式。因为有了下标，我们就可以通过下标来获取或设置这些类型的值，而不需要单独为这些类型编写设置和检索方法。例如，我们可以通过下标来访问数组中的元素，如 somearray［索引］；访问字典中的元素，如 somedictionary［键］。

下标的定义与一般属性的 getter 和 setter 类似，只是要添加“subscript”关键字，并且没有属性名称，如：

```
subscript(index: Int) -> Int {
    get {
      //返回与入参匹配的 Int 类型的值
    }

    set(newValue) {
      //执行赋值操作
    }
}
```

当然，我们也可以设置为类似只读计算属性一样，不能设置值，只能获取值，如：

```
subscript(index: Int) -> Int {
    //返回与入参匹配的 Int 类型的值
}
```

我们创建一个通过下标来返回计算圆面积的程序，如：

```
struct Square{
    //这里定义一个下标方法
```

```
        subscript(radius:Double)->Double{
            //Double.pi：圆周率
            //pow：a 和 b 两个参数，求 a 的 b 次方
            return  Double.pi * pow(radius, 2)
        }
    }

    let 圆的面积 = Square()
    //调用下标
    print("圆圈的面积是：\(圆的面积[10])")
    输出：
    圆圈的面积是：314.159265358979
```

在上述例子中，我们定义了一个结构体“Square”，在里面定义了一个下标方法“subscript”，参数是半径“radius”，里面直接通过内置常量“Double.pi”来获取圆周率，再通过内置函数“pow()”来一起计算出圆的面积，并返回数据。当创建“Square”的实例后，通过实例访问下标“圆的面积[10]”来调用该下标方法，把“10”作为参数传到“subscript”里并返回计算出的面积。

13.2 下标的用法

下标一般在集合类型中使用较多，作为数组、字典访问集合元素和设置值的快捷方式，如：

```
    var array = [1,2,3,4,5,6,7]
    print("数组第 3 个元素是：\(array[2])")
    输出：
    数组第 3 个元素是：3
```

上述例子通过数组下标“[2]”获取的数组第 3 个元素是“3”。

```
    var 学习班 = ["语文科代表": "张三", "数学科代表": "李四", "英语科代表": "王五",
"学习委员": "小明"]
    var 学习委员 = 学习班["学习委员"]!
    print("过去的学习委员：\(学习委员)")
    学习班["学习委员"] = "华仔"
    学习委员 = 学习班["学习委员"]!
    print("现在的学习委员：\(学习委员)")
    输出：
    过去的学习委员：小明
    现在的学习委员：华仔
```

上述例子通过“学习班”字典的下标“[“学习委员”]”来获取的值是“小明”，再通过下标修改值为“华仔”。

关于数组、字典等使用下标的情况，可以参见第 5 章。

13.3　下标脚本

下标脚本允许任意数量的入参索引，并且每个入参类型也没有限制。下标脚本的返回值也可以是任何类型。下标脚本可以使用变量参数和可变参数，但使用写入–读出（in-out）参数或给参数设置默认值都是不允许的。

一个类或结构体可以根据自身需要提供多个下标脚本实现，在定义下标脚本时，通过入参对类型进行区分，使用下标脚本时会自动匹配合适的下标脚本实现运行，这就是下标的重载。

只有一个参数的下标是最常见的情况，但只要有合适的场景，也可以定义多个参数的下标。如下示例，我们定义了一个 Matrix 结构体，将呈现一个 Double 类型的二维矩阵，Matrix 结构体的下标脚本需要两个整型参数：

```
struct Matrix{
    var rows, columns : Int
    var grid: [Int]

    init(rows:Int,columns:Int){
          self.rows = rows
          self.columns = columns

          grid = Array(repeating: 0, count: rows * columns)

    }

    //检查索引是否超越数组大小
    func indexIsValid(row:Int,column:Int)->Bool{
         return (row >= 0 && row < rows && column >= 0 && column < columns)
    }
    //用下标方法来存储矩阵对应的数组
    subscript(row:Int,column:Int)->Int{
        get{
             assert(indexIsValid(row: row, column: column), "数组索引越界")
             return grid[(row * columns) + column]
        }
        set{
            assert(indexIsValid(row: row, column: column), "数组索引越界")
            grid[(row * columns) + column] = newValue
        }
    }
```

```
}

//创建一个 3×3 的矩阵
var matrixdemo = Matrix(rows: 3, columns: 3)
matrixdemo[0,0] = 1
matrixdemo[0,1] = 2
matrixdemo[0,2] = 3
matrixdemo[1,0] = 4
matrixdemo[1,1] = 5
matrixdemo[1,2] = 6
matrixdemo[2,0] = 7
matrixdemo[2,1] = 8
matrixdemo[2,2] = 9

print("矩阵中的项是：")
for item in matrixdemo.grid{
      print("\(item),", terminator: "")
}
```

输出：

矩阵中的项是：

1,2,3,4,5,6,7,8,9,

上述例子中，我们创建了一个矩阵的结构体，初始化参数是最大行数“rows”和最大列数“columns”，创建一个足够容纳 rows * columns 个数的 Int 类型数组。为了存储，将数组的大小和数组每个元素初始值 0，都传入数组的构造方法中来创建一个大小正确的新数组。

我们知道，矩阵中的“元素序号＝所在列 + [（所在行）*总列数]”，为了保证我们传入多参数下标“subscript”方法参数的正确性，我们添加了断言，创建了“indexIsValid”方法来判断参数是否正确，如果参数不正确，则将显示“数组索引越界”。

第 14 章
Swift 继承

本章将讲述 Swift 的继承，包含以下内容。

- 继承的定义
- 继承的一般用法
- 重写父类的方法和属性
- 防止父类的特性被重写

14.1 继承的定义

继承：一个类可以继承另一个类的属性、方法和其他特性（就像我们可以继承自己父母的财产一样）。当一个类继承另一个类时，它称为超类（子类），被继承的类称为基类（父类）。继承是类区别于其他类型的特定行为，简单来说，在 Swift 中，只有类才可以使用“继承”!

在 Swift 中，可以通过继承去调用父类的属性、方法和下标，也可以通过“overide”关键字重写父类的属性、方法和下标。Swift 会帮我们检查重写的属性、方法和下标是否在父类中已经被定义，并检查重写的正确性。

在 Swift 中，可以为类中继承来的属性添加“属性观察器”，这样一来，当属性值改变时，类就会被通知到。可以为任何属性添加“属性观察器”，无论它原本被定义为“存储属性”还是“计算属性”。

14.2 继承的一般用法

首先，我们要有一个父类（基类），然后再定义一个子类（超类）去继承它。

- 父类（基类）：不继承于其他类的类。
- 子类（超类）：在一个已有类的基础上创建的类。

我们一般在创建子类（超类）时，通过“：”符号去继承一个父类（基类）。

我们可以通过“super”关键字去调用父类（基类）的属性、方法、下标：

- 在方法 someMethod 的重写实现中，可以通过“super.someMethod()”来调用父类（基类）的 someMethod 方法。

- 在属性 someProperty 的 getter 或 setter 的重写实现中，可以通过“super.someProperty”来访问父类（基类）的 someProperty 属性。
- 在下标的重写实现中，可以通过“super[someIndex]”来访问父类（基类）的下标。

```
//父类
class 车{
    var 时速 = 0
    var 描述 : String{
        return "时速\(时速)千米每小时"
    }

    func 声音(){
    }
}

//子类继承父类的一般使用
class 自行车: 车{
    var 有篮子 = false
}
let 一辆自行车 = 自行车()
一辆自行车.有篮子 = true
一辆自行车.时速 = 25
print("一辆自行车：\(一辆自行车.描述)，是否有篮子：\(一辆自行车.有篮子)")
输出：
一辆自行车：时速 25 千米每小时，是否有篮子：true
```

在上述例子中，我们定义了子类“自行车”，继承了父类“车”，并添加了属性“有篮子”，然后创建“自行车”的实例“一辆自行车”，就可以调用父类“车”的“时速”属性了，我们设置为“25”，然后在打印输出时，我们继续调用父类的属性“描述”，可以看到相关打印结果，证实我们的子类确实拥有了父类定义的属性。

14.3 重写父类的方法和属性

重写：子类为继承来的属性、方法、下标进行自己需要的改写。

如果要重写父类某个特性，需要在重写的地方前面加上“override”关键字。“override”关键字会提醒编译器去检查该类的父类是否有匹配重写版本的声明。这个检查可以确保我们的重写是正确的。

14.3.1 重写父类的方法

```
class 动车: 车{
```

```
        //重写方法
        override func 声音(){
            print("嘟---嘟---嘟")
        }
    }
    let 一辆动车 = 动车()
    print("一辆动车的声音是：", terminator: "")
    一辆动车.声音()
    输出：
    一辆动车的声音是：嘟---嘟---嘟
```

在上述例子中，我们创建新的子类“动车”继承父类“车”，使用关键字“override”重写了父类的“声音”这个方法，然后通过创建子类的实例“一辆动车”，调用重写的方法“声音”，可以看到输出的结果就是我们子类重写“声音”方法的结果。

14.3.2　重写父类的属性

```
    class 汽车：车{
        var 轮子数 = 0

        //重写为只读 getter
        override var 描述:String{
            //调用父类的方法
            return super.描述 + ",有\(轮子数)个轮子"
        }

        //重写为属性观察器
        override var 时速:Int{
            didSet{
                if(时速 > 120){
                    print("警告：你速度超过 120 千米每小时，为了您的安全，请进入自动驾驶模式！", terminator: "")
                }
            }
        }

    }
    let 一辆汽车 = 汽车()
    一辆汽车.轮子数 = 4
    一辆汽车.时速 = 150
    print(一辆汽车.描述)

    输出：
```

```
警告：你速度超过 120 千米每小时，为了您的安全，请进入自动驾驶模式！时速 150 千米每小时，有 4 个轮子
```

在上述例子中，我们创建子类“汽车”继承父类“车”，通过“override”关键字来重写父类的“描述”属性为只读的 getter 属性，并通过关键字“super”来调用原来父类的“描述”属性。然后再使用“override”关键字来重写父类的“时速”属性为“属性观察器”，如果时速超过 120 则输出警告信息。

14.4 防止父类的特性被重写

我们可以把类、属性、方法、下标在定义前加上关键字“final”，这样就可以防止父类的特性被重写了。

如果我们重写了带有关键字“final”的属性、方法或下标，在编译时会报错。如果我们继承“final”类，编译器也会报错，因为这样的类是不能被继承的。

第 15 章
Swift 初始化

本章将讲述 Swift 的初始化，包含以下内容。

- 初始化的定义
- 为存储属性设置初始值
- 自定义构造函数
- 默认构造函数
- 值类型的初始化
- 引用类型的初始化
- 可失败构造函数
- 必要构造函数
- 通过闭包或方法设置属性的默认值

15.1 初始化的定义

初始化：类、结构体或枚举实例化前的准备过程，此过程主要是为存储属性赋予初始值，并执行创建实例需要的一些值的设置或初始化（简单来说，初始化主要是在创建实例之前进行一些属性的赋值或计算）。

构造函数（构造器）：通过调用自定义构造函数（构造器）去创建实例的初始化过程，并且，构造函数（构造器）是没有返回值的，构造函数（构造器）的主要作用是确保实例在创建前被正确初始化。

15.2 为存储属性设置初始值

类和结构体在创建实例时，必须为所有的存储属性设置合适的初始值。存储属性必须有一个默认值。构造函数在创建某个特定类型的实例时调用，它的最简形式类似于一个不带任何参数的方法，以关键字“init”命名。

当然，我们既可以在构造函数中为存储属性赋初始值，也可以在定义属性时为其设置默认值，如：

```
class 团购{
    //直接为属性赋值
    let 家电:String = "洗衣机"
    let 手机:String
    init(){
        //同构构造函数为属性赋值
       手机 = "苹果手机"
    }
}
let 一起去团购 = 团购()
print("我们一起去团购：\(一起去团购.家电)，还有：\(一起去团购.手机)")
```

输出：

我们一起去团购：洗衣机，还有：苹果手机

在上述例子中，我们的“家电”存储属性，在定义时，直接给予默认值“洗衣机”，然后创建了一个构造函数“init”，里面为存储属性“手机”赋予初始值“苹果手机”，当我们创建类“团购”的实例“一起去团购”时，构造函数“init”就会被调用，并给予存储属性“手机”赋值为“苹果手机”。

15.3 自定义构造函数

我们可以在一个类或结构体里编写一个或一个以上的构造函数，自定义构造函数主要体现在：

- 参数的名称不同；
- 参数的类型不同。

与普通函数一样，构造函数也是可以使用函数别名和外部参数的，如：

```
class 试衣间温度{
     var 正常温度:Double = 26
     init(夏天升温 升温: Double){
           正常温度 += 升温
           print("夏天试衣间温度：\(正常温度)")
     }
     init(冬天降温 降温: Double){
           正常温度 -= 降温
           print("冬天试衣间温度：\(正常温度)")
     }
}
let 冬天试衣间温度 = 试衣间温度(冬天降温: 12)
let 夏天试衣间温度 = 试衣间温度(夏天升温: 11)
```

输出：

冬天试衣间温度：14.0

夏天试衣间温度：37.0

从上述例子可以看到，两个参数的个数和参数类型一样，只是参数名称不一样的构造函数。在调用这两个参数名称不一样的构造函数后，输入相关的值，会在创建实例时为属性“正常温度”赋予不同的值。

15.4　默认构造函数

Swift 2.0 将为所有属性已提供默认值且自身没有定义任何构造函数的结构体或类，提供一个默认的构造函数。这个默认构造函数将简单地创建一个所有属性值都设置为默认值的实例。

虽然类和结构体的默认构造函数都不用我们自己定义，但使用时还是有以下一些区别的。

- 类的默认构造函数：调用时没有任何参数。
- 结构体的默认构造函数：调用时分为没有任何参数的构造函数和逐一显示成员属性参数的构造函数两种。

如：

```
class 购物车{
    var 商品名称: String?
    var 数量 = 0
    var 是否购买 = false
}
let 我的购物车 = 购物车()
print("我的购物车里有商品: \(String(describing: 我的购物车.商品名称))")
```

输出：

我的购物车里有商品：nil

在上述例子中，由于在类“购物车”中没有定义任何自定义构造函数，并且“购物车”是一个父类（基类），所以在创建“购物车”的实例“我的购物车”时，会使用“默认构造函数”为属性赋予自己定义的初始值，虽然“商品名称”没有赋予默认值，但“商品名称”是个可选字符串类型，没有赋值时，它的默认值就是 nil。因此，我们打印出的“商品名称”也是 nil。

我们再来看看结构体的逐一显示属性成员为参数并赋初始值的默认构造函数：

```
struct 试衣间{
    var 镜子: Int
    var 柜子: Int
    var 拖鞋: Int
}
let 优衣库试衣间 = 试衣间(镜子: 1, 柜子: 1, 拖鞋: 2)
print("优衣库试衣间有: 镜子\(优衣库试衣间.镜子)个")
```

```
print("优衣库试衣间有：柜子\(优衣库试衣间.柜子)个")
print("优衣库试衣间有：拖鞋\(优衣库试衣间.拖鞋)个")
输出：
优衣库试衣间有：镜子 1 个
优衣库试衣间有：柜子 1 个
优衣库试衣间有：拖鞋 2 个
```

在上述例子中，结构体“试衣间”没有创建自定义的构造函数，结构体里也没有为属性赋予初始值，所以我们在创建结构体“试衣间”的实例“优衣库试衣间”时，就会调用结构体的“默认构造函数”为结构体里的属性一一赋初始值。

15.5 值类型的初始化

在值类型的初始化中，我们使用比较多的是值类型的“代理构造函数”，因为它能在创建实例时，部分初始化实例的属性和调用多个自己定义的构造函数。也就是说，构造函数调用其他构造函数来完成实例创建时的部分属性初始化过程，我们称之为“构造函数代理”。

在下面例子中，我们定义了一个“学生”结构体，里面有“语文”和“体育”两个结构体来存储这两个科目的“平时分”和“考试分”属性，并提供了初始值 0。

```
struct 语文{
    var 平时分 = 0, 考试分 = 0
}
struct 体育 {
    var 平时分 = 0, 考试分 = 0
}
```

我们可以通过以下三种方式为“考试”创建实例。

- 没有参数的构造函数：默认的 0 值来初始化“语文”和“体育”的存储属性。
- 自定义构造函数：使用特定的“语文成绩”和“体育成绩”来初始化。
- 自定义代理构造函数：使用特定的“特招语文成绩”和“体育成绩”来初始化。

在下面“考试”结构体的定义中，我们为这三种方式提供了三个自定义的构造器：

```
struct 考试 {
    var 语文成绩 = 语文()
    var 体育成绩 = 体育()

    init(){print("这家伙平时逃课，也没有参加考试！")}
    init(语文成绩: 语文, 体育成绩: 体育){
        self.语文成绩 = 语文成绩
        self.体育成绩 = 体育成绩
        print("成绩是：\(self.语文成绩)，\(self.体育成绩)")
    }
```

```
        init(特招语文成绩:语文,体育成绩:体育){
            let 特招语文平时分 = 特招语文成绩.平时分 + 10
            let 特招语文考试分 = 特招语文成绩.考试分 + 10
            self.init(语文成绩: 语文(平时分: 特招语文平时分, 考试分: 特招语文考试分),
体育成绩: 体育成绩)
            print("特招生成绩是: \(self.语文成绩), \(self.体育成绩)")
        }
    }
```

第一个“考试”构造函数“init()”，其功能与默认构造函数是一样的。这个构造函数是一个空函数，使用一对花括号“{}”来描述（我们在这里为了方便测试，提供了一个 print 函数打印相关消息），它没有执行任何属性的赋值和计算等构造过程。调用这个构造器将返回一个新的“考试”实例，它的“语文”和“体育”这两个结构体属性都会使用定义时的默认值“语文（平时分：0，开始分：0）”和“体育（平时分：0，考试分：0）”，如：

```
let 波波考试 = 考试()
输出：
这家伙平时逃课，也没有参加考试！
```

第二个“考试”构造器“init（语文成绩：体育成绩：）”，其功能与结构体在没有自定义构造函数时获得的逐一成员构造函数是一样的。这个构造函数只是简单地将“语文成绩”和“体育成绩”的参数值赋给对应的存储属性，如：

```
let 老镇考试 = 考试(语文成绩: 语文(平时分: 90, 考试分: 80), 体育成绩: 体育(平时分:
80, 考试分: 70))
输出：
成绩是：语文(平时分: 90, 考试分: 80)，体育(平时分: 80, 考试分: 70)
```

第三个“考试”构造函数“init（特招语文成绩：体育成绩：）”看上去复杂一些。（由于这所学校存在着特招生加分项……）先通过特招生的“平时分”加“10”和“考试分”加“10”，然后再调用（或代理给）init（语文成绩：体育成绩：）构造函数将新的“语文成绩”和“体育成绩”值赋值到对应的属性中，如：

```
let 华仔考试 = 考试(特招语文成绩: 语文(平时分: 75, 考试分: 85), 体育成绩: 体育(平时
分: 85, 考试分: 88)
输出：
成绩是：语文(平时分: 85, 考试分: 95)，体育(平时分: 85, 考试分: 88)
特招生成绩是：语文(平时分: 85, 考试分: 95)，体育(平时分: 85, 考试分: 88)
```

我们使用代理构造函数的好处是：函数功能单一性好，可以更好地维护函数功能，减少代码的冗余，简洁明了。

15.6　引用类型的初始化

引用类型的初始化主要是指：类里面的所有存储属性，包括所有继承自父类的属性，都必须在构造过程中设置初始值。

Swift 提供了两种类型的类构造函数来确保所有类在创建实例时，存储属性都能获得初始值，它们分别是指定构造函数和便利构造函数。

15.6.1 指定构造函数和便利构造函数

1. 指定构造函数

- 类中最主要的构造函数。
- 每个类至少有一个指定构造函数。
- 初始化本类中提供的所有属性。
- 调用父类的构造函数初始化父类的属性。
- 语法结构：

```
init(参数) {
      语句
}
```

2. 便利构造函数

- 类中次要的构造函数（辅助用）。
- 可以调用类中的指定构造函数（可以为其属性赋予默认值）。
- 创建特别类型的实例（异类）。
- 是调用指定构造函数的快捷方式。
- 语法结构：

```
convenience init(参数) {
    语句
}
```

15.6.2 类的代理初始化调用

为了简化指定构造器和便利构造器间的调用关系，Swift 采用以下三条规则来限制构造器间的代理调用。

（1）指定构造函数必须调用其直接父类的指定构造函数（向上代理）。

（2）便利构造函数必须调用同一类中定义的其他构造函数（横向代理）。

（3）便利构造器必须最终以调用一个指定构造函数结束（指定结束）。

这些规则的说明如图 15-1 所示。

父类“Superclass”中包含一个指定构造函数“Designated”和两个便利构造函数“Convenience”。其中一个便利构造函数调用了另一个便利构造函数，而后者又调用了唯一的指定构造函数。这满足了上面提到的规则（2）和（3）。这个父类没有自己的父类，所以规则（1）没有用到。

在子类“Subclass”中包含两个指定构造函数“Designated”和一个便利构造函数“Convenience”。便利构造函数必须调用两个指定构造函数中的任意一个，因为它只能调用同一个类里的其他构造函数。这满足了上面提到的规则（2）和（3）。而两个指定构造函数

必须调用父类中唯一的指定构造函数，这满足了规则（1）。

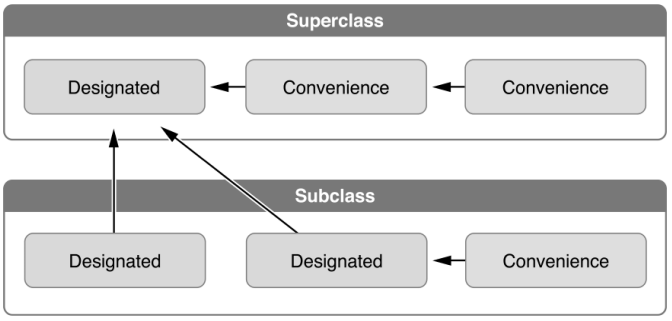


图 15-1

如图 15-2 所示，展示了一种针对四个类的更复杂的类层级结构。它演示了指定构造函数是如何在类层级中充当“管道”的作用，在类的构造器链上简化了类之间的相互关系。

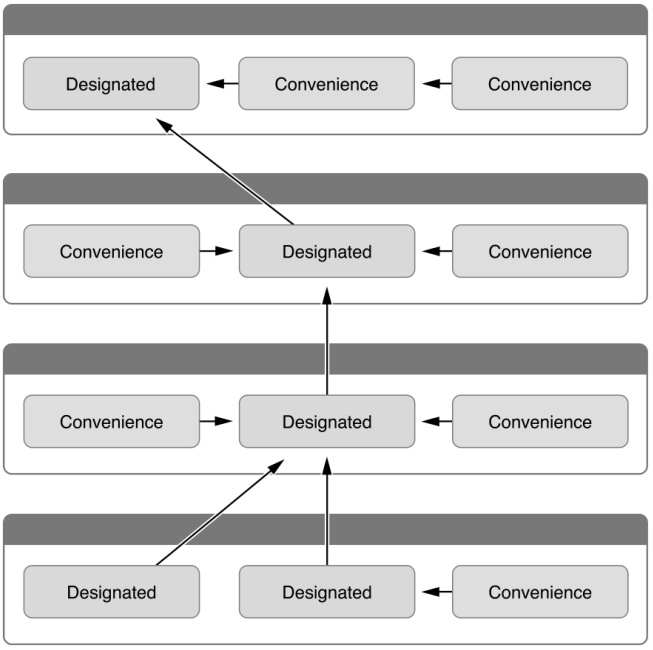


图 15-2

15.6.3　两段式构造过程

Swift 中类的构造过程包含两个阶段。

第一个阶段是每个存储属性通过引入其类的构造函数来设置初始值。当每个存储属性值被确定后，第二个阶段开始，它给每个类一次机会在新实例准备使用前进一步定制其存储型属性。

两段式构造过程的使用，让类在构造过程中更安全，同时在整个类层级结构中给予了每个类完全的灵活性。两段式构造过程可以防止属性值在初始化前被访问，也可以防止属性被另一个构造器意外地赋予不同的值。

Swift 编译器将执行 4 种有效的安全检查，以确保两段式构造过程能顺利完成。

安全检查 1：

指定构造函数必须保证它所在类引入的所有属性都必须先初始化完成，之后才能将其他构造任务向上代理给父类中的构造函数。

如上所述，一个对象的内存只有在其所有存储属性确定后才能完全初始化。为了满足这个规则，指定构造函数必须保证它所在类引入的属性在它往上代理前先完成初始化。

安全检查 2：

指定构造函数必须先向上代理调用父类构造函数，然后再为继承的属性设置新值。如果没有这么做，指定构造函数赋予的新值将被父类中的构造函数所覆盖。

安全检查 3：

便利构造函数必须先代理调用同一类中的其他构造函数，然后再为任意属性赋新值。如果没有这么做，便利构造函数赋予的新值将被同一类中的其他指定构造函数所覆盖。

安全检查 4：

构造函数在第一个阶段构造完成前，不能调用任何实例方法、不能读取任何实例属性的值，self 的值不能被引用。

类的实例在第一个阶段结束以前并不是完全有效的，仅能访问属性和调用方法，一旦完成第一个阶段，该实例才会声明为有效实例。

以下是两段式构造过程中基于上述安全检查的构造流程展示。

阶段 1：

- 某个指定构造函数或便利构造函数被调用。
- 完成新实例内存的分配，但此时内存还没有被初始化。
- 指定构造函数确保其所在类引入的所有存储属性都已赋初值。存储属性所属的内存完成初始化。
- 指定构造函数将调用父类的构造函数，完成父类属性的初始化。
- 这个调用父类构造函数的过程沿着构造函数链一直向上执行，直至构造函数链的最顶部。
- 当到达了构造函数链最顶部，且已确保所有实例包含的存储属性都已经赋值时，这个实例的内存被认为已经完全初始化。此时阶段 1 完成。

阶段 2：

- 从顶部构造函数链一直向下，在每个构造函数链中，类的指定构造函数都有机会进一步定制实例。构造函数此时可以访问 self、修改它的属性并调用实例方法等。
- 最终，任意构造函数链中的便利构造函数可以有机会定制实例和使用 self。

如图 15-3 所示为在假定的子类和父类之间构造的阶段 1。

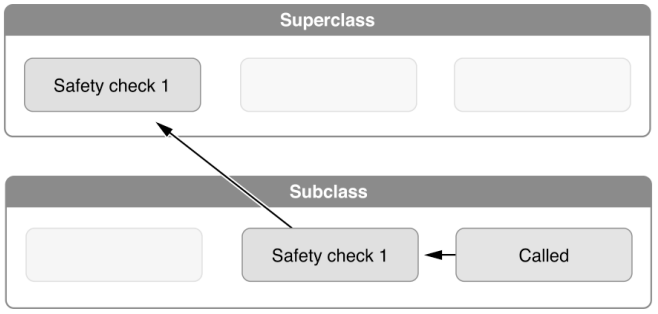


图 15-3

在这个例子中，构造过程从对子类“Subclass”中一个便利构造函数“Called”的调用开始。这个便利构造函数此时无法修改任何属性，它把构造任务代理给同一类中的指定构造函数“Safety check1”。

如安全检查 1 所示，指定构造函数将确保所有子类的属性都有值。然后它将调用父类的指定构造函数，并沿着构造函数链一直向上完成父类的构建过程。

父类中的指定构造函数确保所有父类的属性都有值。由于没有更多的父类需要构建，也就无须继续向上做构建代理。

一旦父类中所有属性都有了初始值，实例的内存被认为是完全初始化，而阶段 1 也已完成。

如图 15-4 所示为相同构造过程的阶段 2。

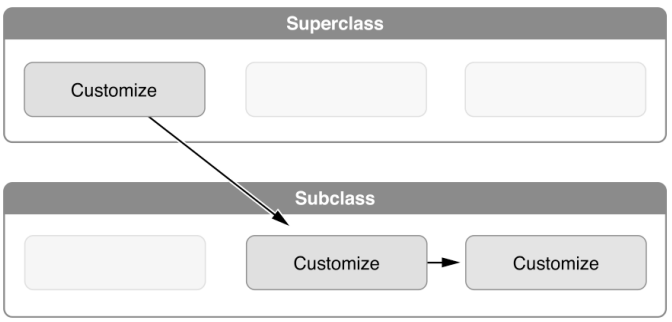


图 15-4

父类中的指定构造函数现在有机会进一步来定制实例（尽管它没有这个必要）。一旦父类中的指定构造函数完成调用，子类的指定构造函数就可以执行更多的定制操作（同样，它也没有这个必要）。

最终，一旦子类的指定构造函数完成调用，最开始被调用的便利构造函数即可执行更多的定制操作。

15.6.4　构造函数的继承和重写

与 Objective-C 中的子类不同，Swift 中的子类不会默认继承父类的构造函数。Swift 的

这种机制可以防止一个父类的简单构造函数被一个更专业的子类继承，并被错误地用来创建子类的实例。

假如我们希望自定义的子类中能实现一个或多个与父类相同的构造函数（也许是为了完成一些自定义的构造过程），我们可以在自定义的子类中提供和重写与父类相同的构造函数。

- 重写指定构造函数：可以在子类里重载它的实现，并在自定义版本的构造函数中调用父类版本的构造函数。
- 重写便利构造函数：我们的重写过程必须通过调用同一类中提供的其他指定构造函数来实现。

例子如下：

```
class 车{
    var 轮子 = 0
    var 描述: String{
        return "\(轮子)个轮子"
    }
}
let 我的车 = 车()
print("我的车轮子：\(我的车.轮子)个")

class 自行车: 车 {
    //重写父类的构造函数
    override init(){
        //调用父类的构造函数
        super.init()
        轮子 = 2
    }
}
let 我的自行车 = 自行车()
print("我的自行车轮子：\(我的自行车.轮子)个")
```

输出：

我的车轮子：0 个
我的自行车轮子：2 个

在上述例子中，我们使用子类“自行车”来继承父类“车”，在子类中重写了父类的构造函数，并在里面再次调用了父类的构造函数，这时，父类的存储属性“轮子”会被赋予默认值“0”，然后在子类的重写函数中再赋予新值“2”。从输出结果中可以看到，父类的存储属性的值已经被我们更改为“2”了。

15.6.5 自动构造函数的继承

如上所述，子类不会默认继承父类的构造函数。但如果特定条件可以满足，父类构造函数是可以被自动继承的。在实践中，这意味着对于许多常见场景，我们不必重写父类的

构造函数，并在尽可能安全的情况下以最小的代价来继承父类的构造函数。

假设要为子类中引入的任意新属性提供默认值，应遵守以下两个规则。

（1）如果子类没有定义任何指定构造函数，它将自动继承所有父类的指定构造函数。

（2）如果子类提供了所有父类指定构造函数的实现（不管是通过规则 1 继承过来的，还是通过自定义实现的），它将自动继承所有父类的便利构造函数。

即使我们在子类中添加了更多的便利构造函数，这两条规则仍然适用。

15.6.6　指定和便利构造器的实战

下面的例子将在实战中展示指定构造函数、便利构造函数和自动构造函数的继承。以下定义了三个类：“食物”、“菜单”及“购物车”的类层次结构，并将演示它们的构造器是如何相互作用的。

第一个类是类层次中的基类，即“食物”，它是一个用来封装食物名字的类。“食物”类引入了一个称为“名称”的 String 类型属性，并提供了一个指定构造函数和一个便利构造函数来创建“食物”实例。

```
class 食物{
    var 名称: String
    //指定构造函数
    init(名称: String){
        self.名称 = 名称
    }
    //便利构造函数
    convenience init(){
        self.init(名称: "没有名称")
    }
}
let 牛肉 = 食物()
print("食物的名称是：\(牛肉.名称)")
输出：
食物的名称是：没有名称
```

在上述例子中，“食物”类中的构造函数“init（名称：String）”被定义为一个指定构造函数，因为它能确保所有新的“食物”实例中所有存储属性都被初始化。“食物”类没有父类，所以“init（名称：String）”构造函数不需要调用“super.init()”来完成构造。

“食物”类同样提供了一个没有参数的便利构造函数“convenice init()”。这个便利构造函数为新的“食物”提供了一个默认参数“没有名称”，通过代理调用同一类中定义的指定构造函数“init（食物：String）”并给参数“食物”传值“没有名称”来实现。

第二个类是“食物”的子类“菜单”。“菜单”类构建了食谱中需要食物的数量。它引入了 Int 类型的存储属性“数量”，以及从“食物”类中继承过来的“名称”属性，并定义了两个构造器来创建“菜单”实例。

```
class 菜单:食物 {
```

```
        var 数量: Int
        //子类的指定构造函数
        init(名称: String, 数量: Int){
            self.数量 = 数量
            super.init(名称: 名称)  //调用父类构造函数
        }
        //子类的便利构造函数
        override convenience init(名称: String) {
            //调用自身的指定构造函数
            self.init(名称: 名称, 数量: 1)
        }
    }
    let 菜单项 = 菜单()
    print("菜单是: \(菜单项.名称) 数量是: \(菜单项.数量)")
    let 炒牛肉 = 菜单(名称: "炒牛肉")
    print("菜单是: \(炒牛肉.名称) 数量是: \(炒牛肉.数量)")
    let 煎鸡蛋 = 菜单(名称: "煎鸡蛋", 数量: 6)
    print("菜单是: \(煎鸡蛋.名称) 数量是: \(煎鸡蛋.数量)")
    输出:
    菜单是：没有名称 数量是：1
    菜单是：炒牛肉 数量是：1
    菜单是：煎鸡蛋 数量是：6
```

在上述例子中，“菜单”类拥有一个指定构造函数“init（名称：String，数量：Int）”，它可以用来产生新“菜单”实例的所有属性值。这个构造函数一开始先将传入的“数量”参数赋值给“数量”属性，这个属性也是唯一在“菜单”中新引入的属性。随后，构造函数将向上代理给父类“食物”的“init（名称：String）”。这个过程满足“两段式构造过程”中的“安全检查 1”。

“菜单”也定义了一个便利构造函数“init（名称：String）”，它只通过“名称”来创建“菜单”的实例。这个便利构造函数假设任意“菜单”实例的“数量”属性为 1，所以不需要显示指明数量即可创建出实例。这个便利构造函数的定义可以让创建实例更加方便和快捷，并避免了使用重复的代码来创建多个“数量”属性为 1 的“菜单”实例。这个便利构造函数只是简单地将任务代理给了同一类里提供的指定构造函数。

注意，“菜单”的便利构造函数“init（名称：String）”使用了与“食物”中指定构造函数“init（名称：String）”相同的参数。因为这个便利构造器要重写父类的指定构造函数“init（名称：String）”，所以必须在前面使用“override”标识。

在这个例子中，“菜单”的父类是“食物”，它有一个便利构造函数“init()”。这个构造函数因此也被“菜单”所继承。这个继承的“init()”函数版本与“食物”提供的版本是一样的，除了它是将任务代理给“菜单”版本的“init（名称：String）”，而不是“食物”提供的版本。

第三个也是最后一个类，即“菜单”的子类“采购车”。这个类构建了采购单中出现的

某一种菜单。

采购单中的每一项总是从“是否购买”属性的“未购买”状态开始的。为了展现这一特性，“采购车”引入了一个布尔类型的属性“是否购买”，它的默认值是“false”。“采购车”还添加了一个计算属性“描述”，它提供了关于“采购车”实例的一些文字描述：

```
class 采购车: 菜单 {
    var 是否购买 = false
    var 描述: String{
        var output = "\(名称): \(数量)"
        output += 是否购买 ? " ✔":" ✘"
        return output
    }
}
```

由于它为自己引入的所有属性都提供了默认值，并且自己没有定义任何构造函数，所以，“采购车”将自动继承所有父类中的指定构造函数和便利构造函数，如：

```
var 早餐列表 = [
    采购车(),
    采购车(名称: "西多士"),
    采购车(名称: "鲜虾云吞", 数量: 8)
]

早餐列表[0].名称 = "鲜榨橙汁"
早餐列表[0].是否购买 = true
for 早餐 in 早餐列表{
    print(早餐.描述)
}
输出:
鲜榨橙汁：1 ✔
西多士：1 ✘
鲜虾云吞：8 ✘
```

在上述例子中，通过字面量方式创建了一个新数组“早餐列表”，它包含三个新的“采购车”实例，因此数组的类型也能自动推导为“采购车[]”。在数组创建完毕后，数组中第一个“采购车”实例的名字从“没有名称”修改为“鲜榨橙汁”，并标记为已购买。接下来通过遍历数组每个元素并打印它们的描述值，展示了所有项当前的默认状态都已按照预期完成了赋值。

15.7　可失败构造函数

如果一个类、结构体或枚举类型的对象，在构造自身的过程中有可能失败，则为其定义一个可失败构造函数，这是非常有必要的。这里所指的“失败”是指：如给构造函数传

入无效的参数值、缺少某种所需的外部资源或不满足某种必要的条件等。

为了妥善处理这种构造过程中可能会失败的情况。我们可以在一个类、结构体或枚举类型的定义中，添加一个或多个可失败构造函数。其语法为在“init”关键字后面添加问号“？”，如“init?”。

注意，可失败构造函数的参数名和参数类型，不能与其他非可失败构造器的参数名及其类型相同。

可失败构造函数在构建对象的过程中，创建一个其自身类型为可选类型的对象。通过“return nil”语句，来表明可失败构造器在何种情况下“失败”。

注意，严格来说，构造函数都不支持返回值。因为构造函数本身的作用，只是为了确保对象自身能被正确构建，所以即使我们在表明可失败构造函数失败的情况下，用到了“return nil”，也不要在表明可失败构造函数成功的情况下，使用关键字“return”，如：

```
struct 动物 {
    let 特别动物 : String
    init?(特别动物: String){
        if 特别动物.isEmpty{
            print("没有输入特别动物")
            return nil
        }
        self.特别动物 = 特别动物
        print("特别动物是：\(self.特别动物)")
    }
}
let 虎狮兽 = 动物(特别动物: "虎狮兽")
let 狮虎兽 = 动物(特别动物: "")
```

在上述例子中，定义了一个名为“动物”的结构体，其中有一个名为“特别动物”的String 类型的常量属性。同时该结构体还定义了一个带一个 String 类型参数“特别动物”的可失败构造函数。这个可失败构造函数被用来检查传入的参数是否为一个空字符串，如果为空字符串，则该可失败构造函数构建对象失败，否则成功。

我们也可以通过构造一个带一个或多个参数的可失败构造函数来获取枚举类型中特定的枚举成员，还可以在参数不满足我们所期望的条件时，导致构造失败，如：

```
enum 工作日{
    case 周一,周二,周三,周四,周五
    init?(星期几: String){
        switch 星期几{
        case "星期一": self = .周一; print("\(星期几)是工作日, \(self)")
        case "星期二": self = .周一; print("\(星期几)是工作日, \(self)")
        case "星期三": self = .周一; print("\(星期几)是工作日, \(self)")
        case "星期四": self = .周一; print("\(星期几)是工作日, \(self)")
        case "星期五": self = .周一; print("\(星期几)是工作日, \(self)")
        default:
```

```
            print("\(星期几)不是工作日！");return nil
        }
    }
}
let 星期六 = 工作日(星期几: "星期六")
let 星期三 = 工作日(星期几: "星期三")
```

输出：

星期六不是工作日！

星期三是工作日，工作日.周一

在上述例子中，定义了一个名为“工作日”的枚举类型。其中包含了五个可能的枚举成员“星期一、星期二、星期三、星期四和星期五”，以及一个被用来找到 String 类型值所对应的枚举成员的可失败构造函数，我们可以通过给该可失败构造器传递合适的参数来获取这三个枚举成员中相匹配的其中一个枚举成员。当参数的值不能与任意一个枚举成员相匹配时，该枚举类型的构建过程失败。

值类型（如结构体或枚举类型）的可失败构造函数，对何时何地触发构造失败这个行为没有任何限制。例如，在前面的例子中，结构体“动物”的可失败构造函数触发失败的行为，甚至发生在“特别动物”属性的值被初始化以前。而对于类而言，就没有那么幸运了。类的可失败构造函数只能在所有的类属性被初始化后和所有类之间的构造函数之间的代理调用发生完后触发失败行为，如：

```
class 产品{
    let 名称: String!
    init?(名称: String){
        self.名称 = 名称
        if 名称.isEmpty{ print("没有产品名称！");return nil}
        print("产品名称：\(self.名称!)")
    }
}
let 皮带 = 产品(名称: "")
let 内衣 = 产品(名称: "内衣")
```

输出：

没有产品名称！

产品名称：内衣

在上述例子中，定义了一个名为“产品”的类，其内部结构与结构体“动物”相似，内部也有一个名为“名称”的 String 类型的属性。由于该属性的值同样不能为空字符串，所以我们加入了可失败构造器来确保该类满足上述条件。但由于“产品”类不是一个结构体，所以当想要在该类中添加可失败构造函数触发失败条件时，必须确保“名称”属性被初始化。因此，我们把“名称”属性的 String 类型做了小小的修改，把其改为隐式解析可选类型（String!），来确保可失败构造函数触发失败条件时，所有类属性都被初始化了。所有可选类型都有一个默认的初始值 nil。因为“名称”属性是一个常量，所以一旦“产品”类构造成功，“名称”属性肯定有一个非 nil 的值。因此，完全可以放心大胆地直接访问“产

品”类的“名称”属性，而不用考虑检查“名称”属性是否有值。

可失败构造函数同样满足在构造函数链中所描述的构造规则。其允许在同一类、结构体和枚举中横向代理其他的可失败构造函数。与之类似，子类的可失败构造函数也能向上代理基类的可失败构造函数。

无论是向上代理还是横向代理，如果代理的可失败构造函数在构造过程中触发了构造失败的行为，整个构造过程都将被立即终止，接下来任何的构造代码都将不会被执行，如：

```
class 篮球:产品 {
    let 数量: Int!
    init?(名称:String, 数量:Int){
        self.数量 = 数量
        super.init(名称:名称)
        if 数量<1 { print("购物篮球数量不能小于 1 个！"); return nil }
        print("你购买的是\(名称)篮球，数量是：\(数量)个！")
    }
}
let SBS 篮球 = 篮球(名称: "", 数量: 0)
let CBA 篮球 = 篮球(名称: "CBA", 数量: 0)
let NBA 篮球 = 篮球(名称: "NBA", 数量: 10)
输出：
没有产品名称！
产品名称：CBA
购物篮球数量不能小于 1 个！
产品名称：NBA
你购买的是 NBA 篮球，数量是：10 个！
```

在上述例子中，定义了一个名为“篮球”的“产品”类的子类。这个类建立了一个在线购物车中的物品的模型，它有一个名为“数量”的常量参数，用来表示该物品的数量至少为 1，与“产品”类中的“名称”属性类似，“篮球”类中的“数量”属性的类型也是一个隐式解析可选类型，只不过由（String!）变为了（Int!）。这样做的目的是为了确保在构造过程中，该属性在被赋予特定的值前能有一个默认的初始值 nil。

可失败构造器总是先向上代理调用基类，“产品”的构造函数“init（名称：）”。这满足了可失败构造器在触发构造失败这个行为前必须总是执行构造代理调用这个条件的。如果由于“名称”的值为空而导致基类的构造函数在构造过程中失败，则整个“篮球”类的构造过程都将失败，后面的子类的构造过程都将不会被执行。如果基类构建成功，则继续运行子类的构造器代码。

如同其他构造函数一样，我们可以用子类的可失败构造函数覆盖基类的可失败构造函数。我们也可以用子类的非可失败构造函数覆盖一个基类的可失败构造函数。这样做的好处是，即使基类的构造函数为可失败构造函数，但当子类的构造函数在构造过程中不可能失败时，我们也可以把它修改过来。

注意，当我们用一个子类的非可失败构造函数覆盖了一个父类的可失败构造函数时，

子类的构造函数将不能再向上代理父类的可失败构造函数。一个非可失败的构造函数永远也不能代理调用一个可失败构造函数，如：

```
class 档案{
    var 文件名: String?
    init(){}
    init?(文件名: String){
        self.文件名 = 文件名
        if 文件名.isEmpty { print("没有写文件名！"); return nil }
    }
}

class X档案: 档案 {
    override init() {
        super.init()
        self.文件名 = "XFile"
        print("文件名是：\(self.文件名!)")
    }
    override init?(文件名: String) {
        super.init()
        if 文件名.isEmpty{
            self.文件名 = "XFile"
        }else{
            self.文件名 = 文件名
        }
        print("文件名是：\(self.文件名!)")
    }
}

let 档案1 = 档案(文件名: "")
let 档案2 = X档案()
let 档案3 = X档案(文件名: "红头文件")
```

输出：

没有写文件名！
文件名是：XFile
文件名是：红头文件

在上述例子中，定义了一个名为“X 档案”的“档案”类的子类。这个子类覆盖了父类的两个指定构造函数，确保了无论在何种情况下，“文件名”属性总有一个非空字符串“XFile”值。

可失败的构造函数的初始化，通过放置一个问号标记“?”在关键字“init”后（init? ），就如创建一个可为“空”的可选类型的实例。当然，我们也可以在定义一个可失败构造函数后，创建一个隐式打开可选类型的实例，即通过放置一个感叹号标记“!”在关键字 init

的后面（init!）代替问号。

- 可以从“init?”使用代理调用“init!”，反之亦然。
- 可以重写“init?”为“Init!”，反之亦然。
- 可以使用“init”使用代理调用“init!”，这样会触发一个“断言”，因为“init!”必定会构造成功，但“init”可能会构造失败。

15.8 必要构造函数

必要构造函数：在父类的构造函数关键字“init”前添加关键字“required”（required init），表明子类重写该构造函数后，通过该函数创建类的实例时，在调用完本类的构造函数后，会再调用父类相关的必要构造函数，如：

```
class 程序员{
    required init(){
        print("每天写代码")
    }
}
class 安卓程序员: 程序员 {
    required init() {
        print("每天写安卓代码")
    }
}

let 男安卓程序员 = 安卓程序员()
输出：
每天写安卓代码
每天写代码
```

在上述例子中，子类“安卓程序员”继承了父类“程序员”，父类创建了一个没有参数的“必要构造函数”，子类也创建了一个没有参数的“必要构造函数”，当子类的实例通过子类的“必要构造函数”创建时，会先打印出子类“必要构造函数”的“每天写安卓代码”，这个函数执行完毕，会继续执行父类的“必要构造函数”，打印出“每天写代码”。

必要构造函数，相当于省略了子类调用父类的需要重写的构造函数（super.init），并只是把关键字“override”改为了“required”。

15.9 通过闭包或方法设置属性的默认值

如果一个存储属性的默认值需要一些定制或设置，我们可以使用一个闭包或全局函数来为该属性提供一个自定义的默认值。每当一个新类型的实例的属性被初始化时，闭包或

函数就会被调用，它的返回值被指定为属性的默认值。

这种类型的闭包或函数一般会创建一个与属性类型相同的临时变量，然后修改它的值以满足预期的初始状态，最后将这个临时变量的值作为属性的默认值返回。

下面列举了闭包如何提供默认值的代码：

```
class SomeClass {
     let someProperty:  a href="" SomeType /a  = {
      //在这个闭包中为 someProperty 创建一个默认值
      //someValue 和 SomeType 是同一类型
               return someValue
      }()
}
```

注意，闭包结尾的花括号后面接了一对空的圆括号“()”。这是用来告诉系统需要立刻执行此闭包。如果忽略了这对圆括号，相当于将闭包本身作为值赋值给了属性，而不是将闭包的返回值赋值给属性。

下面例子定义了一个结构体“Checkerboard”，它构建了西洋跳棋游戏的棋盘，西洋跳棋游戏在一副黑白格交替的 10×10 的棋盘中进行。为了呈现这副游戏棋盘，“Checkerboard”结构体定义了一个属性“boardColors”，它是一个包含 100 个布尔值的数组。数组中的某个元素布尔值为 true 表示对应的是一个黑格，布尔值为 false 表示对应的是一个白格。数组中第一个元素代表棋盘上左上角的格子，最后一个元素代表棋盘上右下角的格子。

“boardColors”数组是通过一个闭包来初始化和组装颜色值的。

```
struct Checkerboard {
     let boardColors: [Bool] = {
          var temporaryBoard = [Bool]()
          var isBlack = false
          for i in 1...10 {
                for j in 1...10 {
                     temporaryBoard.append(isBlack)
                     isBlack = !isBlack
                }
                isBlack = !isBlack
          }
          return temporaryBoard
          }()
     func squareIsBlackAtRow(row: Int, column: Int) -> Bool {
          return boardColors[(row * 10) + column]
     }
}

let board = Checkerboard()
print(board.squareIsBlackAtRow(row: 0, column: 1))
```

```
print(board.squareIsBlackAtRow(row: 9, column: 9))
输出：
true
false
```

输出的图如图 15-5 所示。

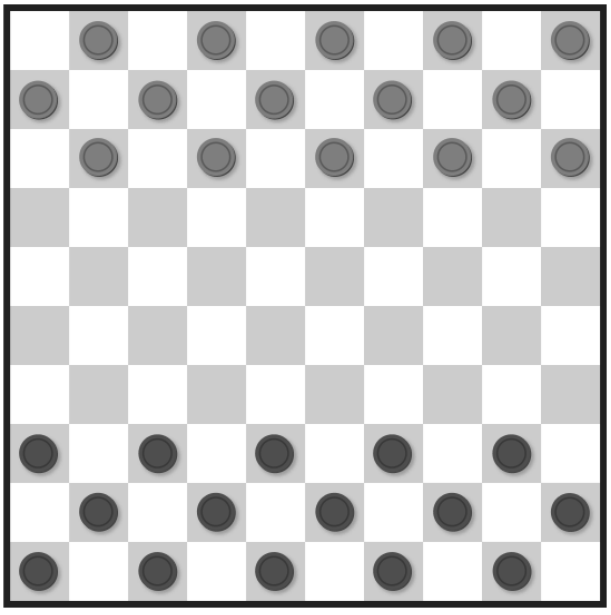


图 15-5

在上述例子中，每当一个新的“Checkerboard”实例创建时，对应的赋值闭包会被执行，一系列颜色值会被计算出来作为默认值赋值给“boardColors”。上例描述的闭包将计算出棋盘中每个格子合适的颜色，将这些颜色值保存到一个临时数组“temporaryBoard”中，并在构建完成时将此数组作为闭包返回值返回。这个返回的值将保存到“boardColors”中，并可以通过“squareIsBlackAtRow”这个工具函数来查询。

第 16 章
Swift 析构

本章将讲述 Swift 的析构，包含以下内容。

- 析构的原理
- 析构的实战

有生，必定有死，有生命的开始，必定有生命的终结；既然我们有实例的诞生（初始化），就必定会有类型的消亡（析构）。

析构就是在一个实例在被内存释放前调用的一个函数，与初始化使用关键字“init”类似。析构我们使用关键字“deinit”，并且，只有类（引用类型）才可以使用析构。

16.1 析构的原理

在 Swift 中，我们一般不用自己释放实例的资源，因为 Swift 2.0 会根据“自动引用计数器”来管理内存。然而，当我们使用一些自己定义资源时，也可以自己进行一些额外的处理。例如，我们创建了一个自定义的类来打开一个文件，并在写入一些数据后，就需要手动在实例被内存释放前关闭该文件，否则，可能会产生错误（简单来说，一般我们不用自己处理内存，系统会搞定）。

- 只有类可以使用析构函数。
- 每个类最多有且只有一个析构函数。
- 析构函数不带任何参数且没有括号。
- 析构函数在实例被内存释放前调用。
- 析构函数不能被别人手动调用。
- 先调用子类的析构函数，再调用父类的析构函数。
- 子类必定会调用父类的析构函数。

16.2 析构的实战

我们先看看一个老虎机和玩家的例子：

```
struct 老虎机 {
```

```
        static var 银币 = 10_000

        static func 赢一场( 赢游戏币: Int)->Int{
            var 赢游戏币 = 赢游戏币
            赢游戏币 = min(赢游戏币 , 银币)
            银币 -= 赢游戏币
            return 赢游戏币
        }
        static func 输光(输游戏币:Int){
            银币 += 输游戏币
        }
    }

    class 玩家 {
        var 游戏币: Int

        init(游戏币: Int){
            self.游戏币 = 游戏币
        }
        func 赢取游戏(赢游戏币: Int){
            游戏币 += 老虎机.赢一场(赢游戏币: 赢游戏币)
        }
        deinit{
            老虎机.输光(输游戏币: 游戏币)
        }
    }
```

首先，我们创建了一个结构体“老虎机”，里面定义了一个静态变量和两个用于计算赢取或输光游戏币的方法。然后，我们创建了一个“玩家”类，里面有存储属性“游戏币”，指定构造函数，用于初始化“游戏币”属性；一个“赢取游戏”的方法，用来获取赢取游戏后，通过调用结构体“老虎机”的“赢一场”方法计算出赢取的“游戏币”。最后，我们还定义了一个析构函数，当“玩家”对象释放内存前，把之前所得的游戏币返还给“老虎机”结构体。

我们先创建一个玩家对象“黄柠檬”，然后初始化给他 100 游戏币，如：

```
var 黄柠檬: 玩家? = 玩家(游戏币: 100)
print("玩家：黄柠檬加入游戏，买了游戏币：\(黄柠檬!.游戏币)")
print("老虎机里现在还有\(老虎机.银币)个银币")
```

输出：

玩家：黄柠檬加入游戏，买了游戏币：100
老虎机里现在还有 10000 个银币

“黄柠檬”运气不错，第 1 盘就赢了 2000 游戏币，如：

```
黄柠檬!.赢取游戏(赢游戏币: 2000)
```

```
print("黄柠檬赢了 2000 游戏币，黄柠檬现在有：\(黄柠檬!.游戏币)")
print("老虎机里现在还剩下\(老虎机.银币)")
```

输出：

黄柠檬赢了 2000 游戏币，黄柠檬现在有：2100
老虎机里现在还剩下 8000

可以看到，我们调用了类的“赢取游戏”方法后，里面调用了“老虎机”的“赢一场”方法，计算出“老虎机”里现有的“银币”和“黄柠檬”赢取了多少游戏币。

最后“黄柠檬”以为自己很牛，结果来个“晒冷”（Show Hand，即一局押注全部游戏币），最后全都输光了。

```
黄柠檬 = nil
print("黄柠檬突然输光，离开游戏")
print("老虎机里这时有：\(老虎机.银币)")
```

输出：

黄柠檬突然输光，离开游戏
老虎机里这时有：10100

可以看到，实例“黄柠檬”等于 nil，这就会在内存释放前调用类定义的析构函数，里面再调用“老虎机”的“输光”函数，把所有银币返回给“老虎机”。

第 17 章
Swift 错误处理

本章将讲述 Swift 的错误处理，包含以下内容。

- 创建错误类型
- 抛出错误
- 捕获和处理错误
- 不处理错误
- 将错误转为可选类型

错误处理是指在我们的程序中产生错误，并且程序快要被内存回收前，我们所进行的一系列处理过程。

一些函数和方法总是不能完全保证可以被完整地执行或有效地输出。可选类型对于不存在的值是非常有用的，但是，当方法产生错误时，错误处理就可以帮助我们的代码做出相应的反应，马上让我们理解是什么原因导致了错误的产生。在 Swift 中，称之为函数或方法的错误抛出。

比如说，我们要做一个读取和处理硬盘上某个文件数据的任务，这存在多种可使任务失败的原因，例如，文件不在指定的路径上、没有读取文件所具有的权限、打开文件时发现文件的编码和我们的程序不兼容等。在这些不同错误的情况下，程序必须能分析相关错误，并从错误中恢复，甚至如果有必要的话，能告知用户错误的原因和情况。

17.1 创建错误类型

在 Swift 中，错误的类型必须来源于“ErrorType”协议。枚举是用来建立属于我们自定义错误类型的最好的方法，我们只要在自己的枚举中添加新的错误类型，并把枚举实现“ErrorType”协议即可，如：

```
enum 自动售货机错误: Error{
    case 不能选择商品
    case 付款金额不够(required: Double)
    case 缺货
}
```

在上述例子中，我们创建了一个“自动售货机错误”的枚举，实现了“Error”接口，

以下情况会产生相关错误。

- 购买的商品，选择了贩卖机里没有的商品。
- 购买的商品，付款比商品的价格低。
- 购买的商品，缺货。

17.2　抛出错误

在抛出错误前，我们需要在函数或方法的返回箭头“-> ”前使用关键字 “throws” 来标明将会抛出错误，如果没有返回值，我们直接在函数参数的结尾右圆括号“)”后添加关键字“throws”，如：

```
func 抛出带返回值错误() throws -> String{
    return "波神错误"
}
func 抛出不带返回值错误() throws{}
```

抛出错误声明后，我们需要在函数或方法里抛出错误，使用关键字“throw”即可，如：

```
struct 商品 {
    var 价格: Double
    var 数量: Int
}

var 库存 = [
    "汉堡": 商品(价格: 15.00, 数量: 7),
    "奶茶": 商品(价格: 10.00, 数量: 10),
    "香蕉": 商品(价格: 5.00, 数量: 11)
]


func 购买(商品名称 名称: String, 购买数量: Int, 付款金额: Double) throws
->Double {
    var 余额 = 0.00

    guard var 库存商品 = 库存[名称] else {
        throw 自动售货机错误.没有此类商品
    }

    guard 库存商品.数量 > 购买数量 else {
        throw 自动售货机错误.商品缺货
    }
```

```
        if 付款金额 >= 库存商品.价格{
            // 正常支付，找钱
            余额 = 付款金额 - 库存商品.价格
            库存商品.数量 = 库存商品.数量 - 购买数量
        }else{
            let 还需要钱 = 库存商品.价格 - 付款金额
            throw 自动售货机错误.付款金额不够(required: 还需要钱)
        }
        return 余额

    }
```

在上述例子中，我们使用了“guard”来进行可选类型值的检查，这是 Swift 提供的一个新的方法，“guard”翻译过来是守护和守卫的意思。在 Swift 中，“guard”和“if”有以下两个重要的区别。

- guard 必须强制使用 else 语句，有 guard 的地方一定有一个 else（一夫一妻）。
- 必须 guard 的条件成立，guard 之后的代码才会运行（我吃后你才能吃）。

在上述例子中，当“库存商品”找不到时，抛出我们之前定义的错误 “自动售货机错误.没有此类商品”；当要购买的商品数量大于“库存商品”数量时，抛出“自动售货机错误.商品缺货”，否则就减少相关“库存商品”数量；当要购买商品的价格大于付款的金额时，抛出“自动售货机错误.付款金额不够”，并计算出还需要多少付款金额，否则，就正常结算，返回余额给用户。

17.3 捕获和处理错误

一般捕获和处理错误，我们采取如下这种结构：

```
do {
    try 调用会抛出错误的函数或方法
    其他执行语句
} catch 匹配自定义错误 {
    处理错误语句
}
```

Swift 中的“do-catch”错误处理机制简单易懂。很多编程语言也使用类似的机制进行错误处理，但在 Swift 中有一个比较重要的特性，即“catch”和“switch”一样具有枚举匹配的能力。因此，使用“catch”可以匹配我们自定义的错误类型，从而可以对错误解析进行更为高级的处理，如：

```
do{
    let 余额 = try 购买(商品名称: "咖啡", 购买数量: 10, 付款金额: 20)
    print("成功购物，余额\(余额)")
```

```
}catch 自动售货机错误.没有此类商品{
    print("没有此类商品！")
}catch 自动售货机错误.商品缺货{
    print("此商品缺货")
}catch 自动售货机错误.付款金额不够(let amountRequired){
    print("要购买此商品，你还需要\(amountRequired)钱")
}
输出：
没有此类商品！
```

在上述例子中，使用了“do-catch”错误处理机制，当调用”购买”方法时，由于我们的库存里并没有“咖啡”这个名称的商品，根据 catch 的捕获机制，能匹配出我们自定义的错误类型是“自动售货机错误.没有此类商品”，然后，我们可以向用户输出“没有此类商品！”。

```
do{
    let 余额 = try 购买(商品名称：“汉堡”, 购买数量：10, 付款金额：30)
    print("成功购物，余额\(余额)")
}catch 自动售货机错误.没有此类商品{
    print("没有此类商品！")
}catch 自动售货机错误.商品缺货{
    print("此商品缺货！")
}catch 自动售货机错误.付款金额不够(let amountRequired){
    print("要购买此商品，你还需要\(amountRequired)钱")
}
输出：
此商品缺货！
```

在上述例子中，使用了“do-catch”错误处理机制，当调用”购买”方法时，由于我们的库存里的“汉堡”只有“7”个，而顾客需要“10”个，根据 catch 的捕获机制，能匹配出我们自定义的错误类型是“自动售货机错误.商品缺货”，然后，我们可以向用户输出“此商品缺货！”。

```
do{
    let 余额 = try 购买(商品名称：“奶茶”, 购买数量：5, 付款金额：5.0)
    print("成功购物，余额\(余额)")
}catch 自动售货机错误.没有此类商品{
    print("没有此类商品！")
}catch 自动售货机错误.商品缺货{
    print("此商品缺货")
}catch 自动售货机错误.付款金额不够(let amountRequired){
    print("要购买此商品，你还需要\(amountRequired)钱")
}

输出：
```

```
要购买此商品，你还需要 5.0 钱
```

在上述例子中，使用了“do-catch”错误处理机制，当调用”购买”方法时，由于我们购买的“奶茶”的付款金额“5.0”小于商品的定价“10.0”，根据 catch 的捕获机制，能匹配出我们自定义的错误类型是“自动售货机错误.付款金额不够”，然后，我们输出计算好的“要购买此商品，你还需要 5.0 钱”。

17.4 不处理错误

如果我们不想处理错误，或者我们非常确定某个方法或函数虽然声明会抛出错误，但我们知道自己在使用时是绝对不会抛出任何错误的。在这种情况下 我们可以在函数或方法的前面直接使用“ try!”，如：

```
let 余额 = try! 购买(商品名称: "香蕉", 购买数量: 5, 付款金额: 40)
print("成功购物，余额\(余额)")
输出：
成功购物，余额 35.0
```

在上述例子中，由于我们早已经知道需要“5”个“香蕉”，知道库存有“11”个，并且我们支付了“40”，比商品的价格高了“35”，绝对不会发生任何错误的情况，所以我们使用了“try!”关键字，最后计算出余额“35.0”，向用户输出。

17.5 将错误转为可选类型

当我们想以同样简洁优雅的错误处理方式去处理相关可能引起错误代码时，可以在函数或方法的前面直接使用“ try?”，如：

```
func 获取余额() -> Double? {
      //如果有错误，执行下一条语句，如果没有错误，返回当前数据
      if let 余额1 = try? 购买(商品名称: "汉堡", 购买数量: 2, 付款金额: 5) { return
余额1 }
      if let 余额2 = try? 购买(商品名称: "奶茶", 购买数量: 3, 付款金额: 20){ return
余额2 }
      return nil
}

let 可选余额 = 获取余额()
if 可选余额!.hashValue > 0
{
      print("成功购买，可选余额为：\(可选余额!)")
}else
```

```
{
      print("你不能购买商品！")
}
输出：
成功购买，可选余额为：10.0
```

在上述例子中，我们先定义了一个返回可选“Double”类型的方法“获取余额()”，里面使用了“if 常量 let = try? 函数() {return 常量}”这样的语句，意思就是如果“try? 函数()”没有错误就返回 let 定义的常量，如果有错误，就执行下一条语句。可以看到，我们输出的结果是“成功购买，可选余额为：10.0”，这说明了第一条语句里有错误，执行第二条语句，第二条语句没有错误，就返回了常量的值。

第 18 章
Swift 泛型

本章将讲述 Swift 的泛型，包含以下内容。

- 泛型能解决的问题
- 泛型函数
- 类型参数
- 命名类型参数
- 泛型类型
- 泛型类型的扩展
- 类型约束
- 关联类型
- 条件语句

泛型：能够在我们定义的函数或方法中编写足够灵活、可重用性强的功能，并可以与任何类型的参数按照我们定义的要求进行工作。使用泛型，我们可以写出避免重复和以更抽象的方式明确表达自己意图的代码。

泛型是 Swift 中最强大的功能之一，在 Swift 的标准库里，很多都是使用泛型构建的。事实上，我们已经使用了很多基于泛型的库，只是自己没有察觉而已。例如，数组和字典类型就是泛型集合。我们既可以创建基于“Int”类型的数组，也可以创建基于“String”类型的数据。同样，我们可以为字典的值存储各种自己所需的类型值。

18.1 泛型能解决的问题

下例是一个标准的非泛型函数“数据转换”，可以转换两个“Int”类型的值。

```
func dataChange( a:inout Int, _ b: inout Int){
    print("转换前的数据是：a = \(a), b = \(b)")
    let 模板 = a
    a = b
    b = 模板
    print("转换后的数据是：a = \(a), b = \(b)")
}
```

```
var a1 = 3
var b1 = 107
dataChange(a: &a1, &b1)
输出：
转换前的数据是：a = 3, b = 107
转换后的数据是：a = 107，b = 3
```

在上述例子中，“dataChange”函数使用了“inout”两个输出参数，并在输出参数“b”前使用了“_”符号，这可以在函数调用时，不显式显示参数别名。最后我们调用该函数，输出相应数据转换前和转换后的结果。

如果我们要转换的不是“Int”类型的数据，而是“String”类型的数据，要如何处理呢？一般的方法是复制一份“dataChange”函数，把参数修改一下参数类型就可以了（程序员就是代码的复制修改），如：

```
func dataChange( a:inout String, _ b: inout String){
    print("转换前的数据是：a = \(a), b = \(b)")
    let 模板 = a
    a = b
    b = 模板
    print("转换后的数据是：a = \(a)，b = \(b)")
}
var a2 = "雨琪"
var b2 = "你好"
dataChange(a: &a2, &b2)
输出：
转换前的数据是：a = 雨琪，b = 你好
转换后的数据是：a = 你好，b = 雨琪
```

在上述代码中，我们只是做了“Int”类型的“dataChange”函数的复制和参数修改，但是，如果我们还要支持“Bool”、“Double”、“Char”等类型呢？是否不断复制 N（N 大于 1）份代码，然后再修改参数类型呢？如果我们的算法不是数据转换，是其他计算的赋值算法，那么我们以后维护时就要改 N（N 大于 1）份代码。这简直是程序员的悲剧，不停“搬砖”的后果。

有什么方法可以解决这个“搬砖”导致的恶果呢？有！那就是“泛型”，我们只要一个“泛型”函数，就能搞定！

18.2　泛型函数

泛型函数：可以使用任何类型于函数中，将上述代码中的“dataChange”的泛型函数改写为“dataChange<T>”，可适用于任何不同类型的参数，如：

```
func DataChange<T>( a: inout T, _ b: inout T){
```

```
        print("泛型转换前的数据是：a = \(a), b = \(b)")
        let 模板 = a
        a = b
        b = 模板
        print("泛型转换后的数据是：a = \(a), b = \(b)")

    }
    var a3 = "希达"
    var b3 = "你好"
    DataChange(a: &a3, &b3)
    var a4 = 250
    var b4 = 1314
    DataChange(a: &a4, &b4)
    输出：
    泛型转换前的数据是：a = 希达，b = 你好
    泛型转换后的数据是：a = 你好，b = 希达
    泛型转换前的数据是：a = 250，b = 1314
    泛型转换后的数据是：a = 1314，b = 250
```

在上述例子中，使用了占位符“T”来代替实际类型（如 Int、String、Double 等）。占位符没有提示“T”必须是怎样的一种类型，但它提示了 a 和 b 必须是同一类型“T”，而不管“T”是什么类型。只有在每次被调用实际传入的参数后，才能决定”T“所代表的类型，简而言之，“T”就是一个占位符。

另外，一个不同之处在于这个泛型函数名后面跟着的占位符“T”是用尖括号括起来的“<T>”。这个尖括号告诉了系统这个“T”是“泛型转换”函数所定义的一个类型。因为 T 是一个占位符，所以系统不会去查找“T”的实际类型。

18.3 类型参数

在上述的“泛型转换函数”例子中，占位符“T”是一种类型参数。

类型参数：指定命名一个占位符，使用一对尖括号括起来“<T>”，并且紧跟在函数后面，如“泛型转换函数<T>”。

一旦指定了一个类型参数，那么它就可以定义这个函数的参数类型（如“泛型转换函数”中的参数 a 和 b），或者作为一个函数的返回类型，或者用作函数主体中的注释类型。在这种情况下，被类型参数所代替的占位符，不管函数在任何时候被调用，都会被实际类型所替换（在上述“泛型转换函数”的例子中，当函数第一次被调用时，“T”被“String”类型替换，第二次被调用时，被“Int”类型替换）。并且，我们可以在泛型函数中使用多个类型参数，命名在尖括号中，用逗号分开，如“泛型转换函数<T，T，T>”。

18.4　命名类型参数

通常情况下，类型参数都是使用一些有特别含义的名字来命名的，如在字典“Dictionary<Key，Value>”中，我们使用“Key”和“Value”来代表泛型的类型参数，在数组“Array<Element>”中，我们使用“Element”来代表泛型的类型参数。因为这些参数的命名都与函数有一定的关联，这样可以让调用的人容易理解参数的含义。当然，还有一些没有意义的参数类型的命名，如我们在“泛型转换函数”中使用的“T”占位符，还有两个占位符“U”和“V”都是我们使用无意义类型参数所常用的命名方式。

注意，要始终使用以大写字母开头的驼峰式命名法（如“T”和“KeyType”）来给类型参数命名，以表明它们是类型的占位符，而非类型值。

18.5　泛型类型

除了泛型函数，Swift 可以让我们快速定义自己需要的泛型类型。这些自定义的类、结构、枚举都可以与任何类型进行工作，就像数组和字典一样。

本节将展示如何编写一个泛型集合类型“泛型栈”。“栈”在其他编程语言中就是有序值的集合，类似数组，但又区别于 Swift 的数组。数组允许在任何位置添加和删除项，然而，栈只允许新的项添加到栈的最后位置（把一个值压入栈中），与此类似，从栈中可以删除集合最后的一项（从栈中弹出一个值）。

如图 18-1 所示为一个栈的压入“push”和弹出“pop”行为。

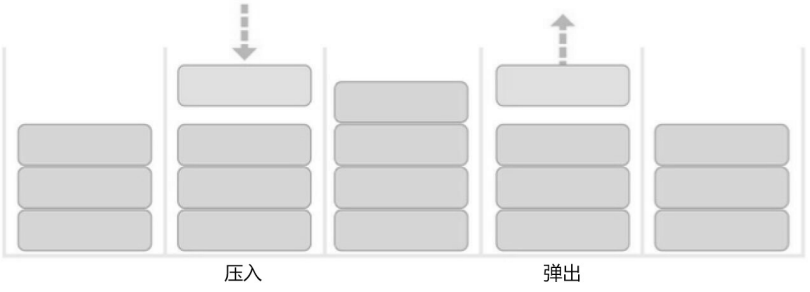


图 18-1

图 18-1 展示了栈的新增项“压入”和删除最后项“弹出”的行为。

（1）目前栈上有 3 个值。

（2）第 4 个值被“压入”栈的最高处。

（3）现在栈中有 4 个值，其中刚被“压入”的值在顶部。

（4）栈里顶部的值被删除或称被“弹出”。

（5）“弹出”顶部的值后，栈只有 3 个值了。

如果栈只用于 Int 类型的值，我们可以编写一个非泛型的函数：

```
struct Int 栈{
    var 数组项 = [Int]()

    mutating func 压入(项:Int){
        数组项.append(项)
    }
    mutating func 弹出() -> Int {
        return 数组项.removeLast()
    }
}
```

在上述例子中，我们定义了一个“Int 栈”，专门处理 Int 类型栈的数据“压入”和“弹出”。我们使用了一个 Int 类型的数组来存储 Int 类型数据，提供了“压入”和“弹出”两个方法向栈中插入一个值和删除一个值。在结构体中，我们必须使用“mutation”关键字来标记这个方法是变异方法，因为这些方法会改变结构体里“数组项”的值。

上例只是展示了“Int 栈”只能用于 Int 类型的值，我们可以将其修改为适用于任何类型的“泛型栈”，如：

```
struct 栈<Element> {
     var 数组项 = [Element]()

     mutating func 压入(项:Element){
         print("压入栈的有：\(项)")
         数组项.append(项)
     }
     mutating func 弹出() -> Element{
         return 数组项.removeLast()
     }
}
```

在上述例子中，可以看到这个版本的“泛型栈”和非泛型版本的“栈”的代码结构相似，只是把类型参数从“Int”类型参数改为了“<Element>”泛型类型参数。泛型参数“Element”在本例的泛型结构体的以下情况中必须调用。

- 当创建一个命名为“数组项”的泛型数组时，必须使用泛型数组定义为“[Element]”。
- 当指定泛型“数组项”中的某一“项”的类型时，必须使用泛型定义为“Element”。
- 当函数返回值是“数组项”中的某一“项”的类型时，必须返回泛型类型“Element”。

当我们初始化这个泛型结构体“栈<Element>”时，必须指定“Element”的类型，如：

```
var 花川学院栈 = 栈<String>()
花川学院栈.压入(项: "波波")
花川学院栈.压入(项: "老镇")
```

```
花川学院栈.压入(项：  "雨琪")
花川学院栈.压入(项：  "华仔")
输出：
压入栈的有：波波
压入栈的有：老镇
压入栈的有：雨琪
压入栈的有：华仔
```

如图 18-2 所示为“泛型栈”如何“压入”这 4 个值到栈里的过程。

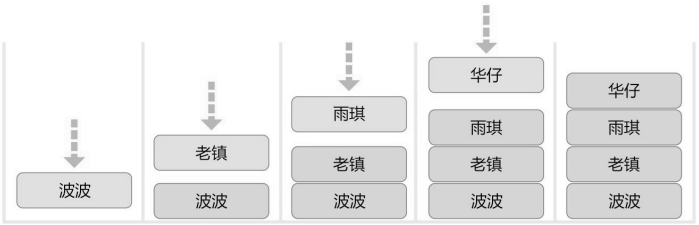


图 18-2

之后，我们从“泛型栈”中“弹出”并删除“华仔”，如：

```
print("花川学院毕业的有：\(花川学院栈.弹出())")
输出：
花川学院毕业的有：华仔
```

如图 18-3 所示为“泛型栈”如何“弹出”值的过程。

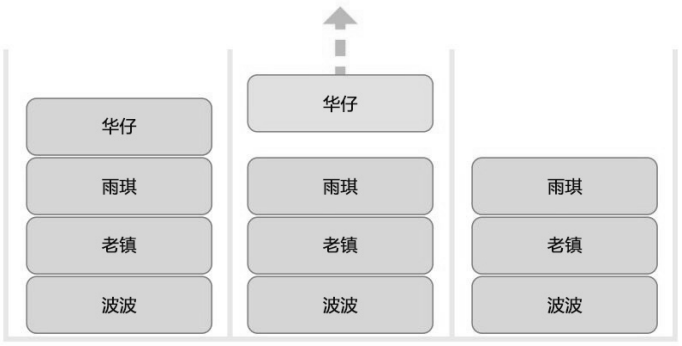


图 18-3

18.6　泛型类型的扩展

当我们想扩展一个泛型类型时，不必提供为扩展泛型提供额外的参数或参列表，相反，可以使用原来“泛型栈”里定义的参数作为扩展的参数使用。

下面的例子将“泛型栈”添加一个只读计算属性“第一项”，它将返回在栈中“压入”

栈的第一项，如：

```
extension 栈{
    var 第一项: Element?{
        return 数组项.isEmpty ? nil : 数组项[数组项.count - 1]
    }
}

var 扩展花川学院栈 = 栈<String>()
扩展花川学院栈.压入(项: "希达")
扩展花川学院栈.压入(项: "小吴")
扩展花川学院栈.压入(项: "书霸")
print("扩展花川学院第一人是：\(扩展花川学院栈.第一项!)")
输出：
扩展花川学院第一人是：书霸
```

在上述例子中，我们的只读计算属性“第一项”还引用了原来的泛型参数类型“Element”作为参数，并提供了可选属性“?”。因为返回的 Element 可能为 nil，所以我们通过原来定义的“数组项”属性，当“数组项”没数据时返回 nil，否则返回我们“数组项”的最后一项作为“第一项”元素（栈的先进后出原理）。

18.7 类型约束

我们在上述“泛型数据转换”和“泛型栈”的例子中，函数的参数都是可以使用任意类型来运行的，然而，当我们只想它们只能使用某些特定的参数类型时，就需要使用泛型函数和泛型类型的“类型约束”加以限制。类型约束：制定一个参数类型必须来源于指定继承的类或实现某个协议或多个协议。例如，Swift 中的字典，约束字典中键的类型必须是一个哈希类型，字典必须通过这个哈希值作为键，才可以被唯一表示，并且可以检查其是否已经包含了某个特定键的值。那么，我们可以为字典加上哪些类型约束吗？当然可以，我们要加的类型约束必须遵循字典的相关协议（Swift 标准库中定义的一个特定协议），幸好，所有 Swift 中的基本类型（String、Int、Double 和 Bool）都是可哈希的，也就是说，这些基本类型都可以作为字典的约束类型。

1. 类型约束语法

我们可以在一个类型参数名后面写一个类型约束，用冒号分割，作为类型参数链的一部分。这种作用于泛型函数的类型约束的基础语法如下（和泛型类型的语法相同）。

```
func someFunction<T: SomeClass, U: SomeProtocol>(someT:  T, someU: U) {
    // 函数的功能写在这里
}
```

在上面这个语法结构中，第一个参数类型“T”需要“T”必须是“SomeClass”的子类为约束类型；第二个参数类型“U”需要“U”必须是实现“SomeProtocol”协议的。

2. 类型约束实战

```
func 查询字符串索引(数组:[String], _ 查询值: String) -> Int? {
    for (索引, 值) in 数组.enumerated(){
        if 值 == 查询值{
            return 索引
        }
    }
    return nil
}
let 动物园 = ["猫","狗","松鼠","鸭子","兔子"]
if let 找到的索引 = 查询字符串索引(数组:动物园, "松鼠"){
   print("动物园里松鼠的索引是：\(找到的索引)")
}
```

输出：

动物园里松鼠的索引是：2

在上述例子中，我们创建了一个非泛型的函数“查询字符串索引”。在函数中，我们通过比较字符串数组的“值”是否全等于“==”参数“查询值”来判断，如果全等于，就返回该字符串数组值的索引，如果不全等于，则返回 nil。

这里我们只针对字符串数组来实现，如果要加入的是整型数组或浮点型数组，那么我们如何实现这个需求呢？使用泛型：

```
func 查询索引<T>(数组:[T], _ 查询值: T) -> Int? {
    for (索引, 值) in 数组.enumerated(){
        if 值 == 查询值{
            return 索引
        }
    }
    return nil
}
```

泛型，不就是加个<T>，把参数类型改成 T 即可吗？为什么这里“值 == 查询值”在编译器中一直报错：

```
binary operator '==' cannot be applied to two T operands
        if 值 == 查询值{
```

这是什么原因呢？原来，不是所有的类型都可以用全等号“==”来进行比较的，例如，我们自己定义的结构体或类，就不能随便用来“全等于”某个值，因为在 Swift 中，无法猜到这个类或结构体是否知道“全等于”的意思。正因如此，编译器就会报错。那么，如果我们把上述例子修改为 Swift 可用的泛型函数呢？其实，在 Swift 的标准库中，已经定义了“Equatable”协议，这个协议要求任何类型都要实现全等操作符“==”和不等操作符“!=”。并且，所有的 Swift 中的基础类型，都已经实现了“Equtable”协议，我们可以这样修改：

```
func 查询索引<T: Equatable>(数组:[T], _ 查询值: T) -> Int? {
```

```
        for (索引, 值) in 数组.enumerated(){
            if 值 == 查询值{
                return 索引
            }
        }
        return nil
    }
    let 浮点数组里的索引 = 查询索引(数组:[1239.3, 0.1, 0.25], 9.3)
    print("浮点数组里 9.3 的索引是：\(String(describing: 浮点数组里的索引))")
    let 字符数组里的索引 = 查询索引(数组:["牛奶","牛奶饼干","芝士面白"], "牛奶饼干")
    print("字符串数组里牛奶饼干的索引是：\(String(describing: 字符数组里的索引!))")
```

输出：

浮点数组里 9.3 的索引是：nil

字符串数组里牛奶饼干的索引是：1

在上述例子中，我们只是把“<T>”改为了“<T：Equatble>”，即要求任何类型都必须实现“Equtable”协议，就可以把泛型类型对应的全等操作符“==”都实现，也就不会再报错了。

18.8 关联类型

当定义一个协议时，有时声明一个或多个关联类型作为协议定义的一部分是非常有用的。一个关联类型作为协议的一部分，给定了类型的一个占位符（或别名）。作用于关联类型上实际类型在协议被实现前是不需要指定的。关联类型被指定为“typealias”关键字，如：

```
protocol 容器 {
    typealias 项的类型
    mutating func 添加(项: 项的类型)
    var 数量: Int { get }
    subscript(i: Int) -> 项的类型 { get }
}
```

“容器”这个协议定义了一个“关联类型”为“项的类型”，并要求实现“容器”这个协议：

- 必须实现“添加”方法，把“项”添加到一个新的容器里；
- 必须实现“数量”这个计算属性，返回容器里的“项”的整型数量；
- 必须实现容器通过整型索引为参数的“下标”（subscript）方法返回相关“项”。

下面我们先看看非泛型的“整型栈”实现“容器”协议是如何实现的：

```
struct 整型栈: 容器 {
    //原有属性和方法
    var 数组项 = [Int]()
    mutating func 压入(项: Int){
        数组项.append(项)
```

```
        }
        mutating func 弹出()->Int{
            return  数组项.removeLast()
        }

        //实现协议
        typealias 项的类型 = Int
        mutating func 添加(项: 项的类型) {
            self.压入(项: 项)
        }
        var 数量: Int{
            return 数组项.count
        }
        subscript(i:Int) ->Int{
            return 数组项[i]
        }
    }
```

在上述例子中，我们创建了一个“整型栈”实现了“容器”指定必须实现的“添加”方法，计算属性“数量”和下标，并把关联类型“项的类型”定义为整型。

同样，参照这个非泛型的“整型栈”，我们可以写出“关联栈”的实现：

```
    struct 关联栈<Element>: 容器 {
        //原有属性和方法
        var 数组项 = [Element]()
        mutating func 压入(项: Element){
            数组项.append(项)
        }
        mutating func 弹出()->Element{
            return  数组项.removeLast()
        }

        //实现协议
        mutating func 添加(项: Element) {
            self.压入(项: 项)
        }
        var 数量: Int{
            return 数组项.count
        }
        subscript(i:Int) ->Element{
            return 数组项[i]
        }
```

```
}
```

在上述例子中，占位符类型参数“Element”是“添加”方法中的“项”的类型，以及下标的返回类型，因为 Swift 可以通过类型推断出“容器”的“关联类型”，即“Element”的合适类型。

除自定义类型可以使用关联类型外，我们也可以指定 Swift 中存在的类型扩展其关联类型，如 Swift 中数组已经提供了下标的方法，我们对数组扩展实现“容器”协议，只要将“增加”方法和计算属性“数量”实现就可以了，如：

```
extension Array: 容器{
    //实现协议
    mutating func 添加(项: Element) {
        self.append(项)
    }
    var 数量: Int{
        return self.count
    }

}
```

18.9 条件语句

约束类型，就是如何对类型进行约束的描述，使我们定义的泛型函数或类型的参数类型符合我们的相关要求。

对关联类型定义约束是非常有用的，我们可以在参数列表中通过“where”语句定义参数的约束。

一个“where”语句能够使一个关联类型遵循一个特定的协议，以及（或）那个特定的参数类型和关联类型可以是相同的。我们可以写一个“where”语句，紧跟在参数类列表后面，“where”语句后跟一个或多个针对关联类型的约束，以及（或）一个或多个类型和关联类型间的等价关系。

下面的例子定义了一个名为“所有匹配项”的泛型函数，用来检查两个“容器”实例是否包含相同顺序的相同元素。如果所有的元素能够匹配，那么返回 true，否则返回 false。被检查的两个“容器”可以不是相同类型的容器（虽然它们可以是），但它们确实拥有相同类型的元素。这个需求通过一个约束类型和“where”语句结合来表示。

```
func 所有匹配项<
    C1: 容器, C2: 容器>
    (第一个容器: C1, 第二个容器: C2) -> Bool
    where C1.项的类型 == C2.项的类型, C1.项的类型: Equatable {

        // 检查两个 Container 的元素个数是否相同
```

```
        if 第一个容器.数量 != 第二个容器.数量 {
            return false
        }

        // 检查两个 Container 相应位置的元素彼此是否相等
        for i in 0..<第一个容器.数量 {
            if 第一个容器[i] != 第二个容器[i] {
                return false
            }
        }

        // 如果所有元素检查都相同，则返回 true
        return true

}
```

在上述例子中，我们定义了一个泛型函数“所有匹配项”，它有两个参数：“第一个容器”参数类型是“C1”和“第二个容器”参数类型是“C2”。

- “C1”必须实现“容器”的协议（C1：容器）。
- “C2”必须实现“容器”的协议（C1：容器）。
- “C1”里“项的类型”必须和“C2”里“项的类型”一致（C1.项的类型 == C2.项的类型）。
- “C1”里“项的类型”必须实现“Equatable”协议（C1.项的类型：Equatable）。

我们调用一下这个函数：

```
var 字符栈 = 关联栈<String>()
字符栈.压入(项: "波波")
字符栈.压入(项: "老镇")
字符栈.压入(项: "华仔")
var 字符数组 = ["波波", "老镇", "华仔"]

if 所有匹配项(第一个容器: 字符栈, 第二个容器: 字符数组){
    print("所有项是一样的！")
}else{
    print("所有项是不一样的！")
}
输出：
所有项是一样的！
```

上例创建了一个“字符栈”来存储字符串，然后往这个“关联栈”里压入了三个字符串。之后再创建一个字符串数组，赋予数组里和“字符栈”里一样的三个字符串。即使“字符栈”和“字符数组”是不同的类型，但它们都实现“容器”的协议，而且它们都包含同样类型的值。因此，我们可以调用“所有匹配项”函数，用“字符栈”和“字符数组”作为它的参数去运行，从输出结果可以看到，它们里的所有项是一样的。

第 19 章
Swift 扩展

本章将讲述 Swift 的扩展，包含以下内容。

- 扩展的定义和语法
- 扩展计算属性
- 扩展构造函数
- 扩展方法
- 扩展下标
- 扩展嵌套函数

19.1 扩展的定义和语法

扩展就是为已存在的类、结构体、枚举、协议等类型添加新的功能，包括在我们没有获得源代码或没有权限修改源代码的情况下为这些类型扩展新功能的能力（即我们不能逆向工程）。因此，扩展只能为各种类型添加新功能，而不能重写或修改本来的功能。

Swif 中的扩展可以：

- 添加计算属性和静态计算属性；
- 定义实例方法和类型方法；
- 提供新的构造函数；
- 定义下标；
- 定义和使用新的嵌套类型；
- 使一个已存在的类型去实现协议。

声明一个扩展，我们使用关键字“extension”在类型定义的前面，如：

```
extension  SomeType{
    // SomeType 添加的新功能写在这里
}
```

19.2 扩展计算属性

扩展可以向已有的类型里添加计算实例属性和计算静态属性。在下面例子中，我们为

“Double”类型添加了 5 个计算实例属性，用来把不同单位的数据都转换为“米”为单位：

```
extension Double{
    var 千米: Double {return self * 1_000.0}
    var 米: Double {return self}
    var 厘米: Double {return self/100.0}
    var 毫米: Double {return self/1_000.0}

}

let 姚明身高 = 226.0.厘米
print("姚明身高是：\(姚明身高)米")
let 每天跑步 = 3.5.千米
print("老镇为了减肥，每天跑步：\(每天跑步)米")
// 走路 150 米 + 坐车 7.5 千米
let 上班距离 = 150.米 + 7.5.千米
print("华仔从家里到公司距离是：\(上班距离)米")
输出：
姚明身高是：2.26 米
老镇为了减肥，每天跑步：3500.0 米
华仔从家里到公司距离是：7650.0 米
```

在上述例子中，我们扩展了“Double”的 5 个计算属性，把“千米”、“厘米”、“毫米”为单位的计算属性转换为“米”。可以看到，我们在扩展中使用了关键字“Self”，这里的“Self”代表“Double”本身。当我们使用扩展后，就能统一单位了，以方便计算出“上班距离”这个“米”和“千米”相加后是多少米的结果。

注意，我们只能扩展计算属性，不能扩展存储属性和观察属性。

19.3　扩展构造函数

扩展可以为存在的类型添加新的构造函数，但它只能是便利构造函数，不能为已存在的类型中添加新的指定构造函数或析构函数；指定构造函数和析构函数只能在原始类型里来提供（只能扩展便利构造函数）。

首先，我们定义一个“角色”结构体和两个辅助结构体“武器”和“防具”，默认值都是 0，如：

```
struct  武器 {
    var 攻击力 = 0, 命中率 = 0
}
struct 防具 {
    var 防御力 = 0, 闪避率 = 0
```

```
    }
    struct 角色 {
        var 装备武器 = 武器()
        var 装备防具 = 防具()
    }
```

然后，我们创建两个“角色”结构体的实例，一个调用其默认的无参数构造函数，另一个调用其默认的逐一属性为参数的构造函数，如：

```
    let 西门吹水 = 角色()
    print("西门吹水：\(西门吹水)")
    let 小李飞镖 = 角色(装备武器：武器(攻击力：100，命中率：300)，装备防具：防具(防御力：200，闪避率：400))
    print("小李飞镖：\(小李飞镖)")
    输出：
    西门吹水：角色(装备武器：__lldb_expr_126.武器(攻击力：0，命中率：0)，装备防具：__lldb_expr_126.防具(防御力：0，闪避率：0))
    小李飞镖：角色(装备武器：__lldb_expr_126.武器(攻击力：100，命中率：300)，装备防具：__lldb_expr_126.防具(防御力：200，闪避率：400))
```

最后，我们为结构体“角色”扩展其便利勾走函数，如：

```
    extension 角色{
        init(带上武器：武器，穿上防具：防具){
            let 实际攻击力 = 带上武器.攻击力 + (穿上防具.防御力 / 10)
            let 实际命中率 = 带上武器.命中率 + (穿上防具.闪避率 / 10)
            self.init(装备武器：武器(攻击力：实际攻击力，命中率：实际命中率)，装备防具：穿上防具)
        }
    }
    let 孤独求败 = 角色(带上武器：武器(攻击力：100，命中率：300)，穿上防具：防具(防御力：200，闪避率：400))
    print("孤独求败：\(孤独求败)")
    输出：
    孤独求败：角色(装备武器：__lldb_expr_126.武器(攻击力：120，命中率：340)，装备防具：__lldb_expr_126.防具(防御力：200，闪避率：400))
```

在上述例子中，我们扩展了“角色”结构体，增加了一个便利构造函数，专门为“人民币玩家”服务，在里面的便利构造函数中，我们重新计算了“角色”的“实际攻击力”和“实际命中率”，然后再通过调用自身“self”的指定构造函数“init(装备武器:武器，装备防具:防具)”来实现对“角色”的构造函数的扩展。

可以看到，创建的“孤独求败”和之前的“小李飞镖”所调用的构造函数不一样，虽然输入的参数是一样的，但“孤独求败”调用的是扩展的便利构造函数，重新计算了相关“角色”的武器的攻击力和命中率。

19.4　扩展方法

扩展方法：可以在已存在的类型中添加新的实例方法和变异方法。在下面的例子中，我们为已存在的类型“Int”整型添加一个“重复打印”的实例方法。

```
extension Int{
    //闭包
    func 重复打印(打印: () -> Void){
        for_in 0..<self{
            打印()
        }
    }
}
3.重复打印(打印: { print("希达猫") })
3.重复打印{print("一起学 Swift！") }
输出：
希达猫
希达猫
希达猫
一起学 Swift！
一起学 Swift！
一起学 Swift！
```

在上述例子中，我们为“Int”扩展了一个实例方法“重复打印”，其参数为一个“闭包”，然后“Int”实例就可以调用“打印”这个方法很多次。可以看到，我们调用“3.重复打印({ print(“希达猫”) })”就打印了 3 次“希达猫”，这个扩展的函数也可以使用挂尾闭包的方法调用，如“3.重复打印{ print(“一起学 swift！”) }”。

当然，有时我们在结构体或枚举中，如果想通过扩展去修改自身“self”或其他属性，则必须在要扩展的方法前添加关键字“mutating”，如：

```
extension Int{
    mutating func square(){
        print("\(self)的平方是：", terminator: "")
        self = self * self
        print("\(self)")
    }
}
var 三 = 3
三.square()
输出：
3 的平方是：9
```

在上述例子中，我们扩展了“Int”，为其添加了一个“square”，计算自身整型数字的平方是多少的函数，因为在结构体里重新计算了自身“self”，所以我们扩展函数时，添加了关键字“mutating”。

19.5 扩展下标

扩展下标：可以为已存在的类型添加新的下标，下面的这个例子就是为整型“Int”添加新的下标。通过下标“[n]”可以计算出“实际工资”。

```
extension Int{
    subscript(绩效: Int) ->Int {
        var 绩效工资 = 绩效 > 0 ? 1000 : 0
        var 我的绩效 = 绩效
        while 我的绩效 > 0 {
            绩效工资 += 100
            我的绩效 -= 1
        }
        return self + 绩效工资
    }
}
print("小波 8 月份工资：\(20000[1])")
print("老镇 8 月份工资：\(10000[0])")
print("雨琪 8 月份工资：\(15000[3])")
输出：
小波 8 月份工资：21100
老镇 8 月份工资：10000
雨琪 8 月份工资：16300
```

19.6 扩展嵌套函数

扩展嵌套函数：可以为已存在的类、结构体和枚举添加新的嵌套类型。

```
extension Int{
    enum 数字枚举{
        case 负数,零,正数
    }

    var 判断数字：数字枚举{
        switch self{
            case 0: return .零
```

```
            case let x where x > 0: return .正数
            default: return .负数
        }
    }
}
```

在上述代码中，我们给整型“Int”扩展了一个嵌套枚举类型“数字枚举”，并添加了只读计算属性“判断数字”，用于判断数字为正数时，返回“数字枚举.正数”，负数时，返回“数字枚举.负数”，零时，返回“数字枚举.零”。

我们添加一个方法，使用整型数组来调用，如：

```
func 打印判断数字的符号(数组: [Int]){
    for 数字 in 数组{
        switch 数字.判断数字{
            case .负数: print("- ", terminator: "")
            case .零: print("0 ", terminator: "")
            case .正数: print("+ ", terminator: "")
        }
    }
}
打印判断数字的符号(数组:[3, 7, -24, 0, -8, 0, 11])
输出:
+ + - 0 - 0 +
```

在上述例子中，我们创建了“打印判断数字的符号”的方法，参数是整型数组“数组:[Int]”，然后在方法里我们遍历“数组”的“数字”，通过我们的扩展方法“判断数字”，然后打印出相关信息。

第 20 章
Swift 协议

本章将讲述Swift的协议，包含以下内容。

- 协议的定义和语法
- 对属性的规定
- 对方法的规定
- 对变异方法的规定
- 对构造函数的规定
- 协议也是一种类型
- 代理
- 在扩展中添加协议
- 协议类型的集合
- 协议的继承
- 协议类型的组合
- 检查协议的一致性

20.1 协议的定义和语法

协议：类似于Java、C#等语言的接口，它通过定义一些列的属性、方法来描绘一个特定任务或功能的一个蓝图（只定义了要求，没具体执行的细节）。协议必须通过某个类、结构体或枚举去实现我们描绘的蓝图。我们将满足协议里的各种需求的具体实现称为“实现协议”，具有“实现协议”的类、结构体或枚举，我们称为协议的“实现类型”。一般协议的“实现类型”是类或结构体。

我们使用关键字“protocol”定义一个协议：

```
    protocol SomeProtocol {
    //协议的内容
}
```

实现协议和继承的语法都是一样，在类或结构体定义的名字后，使用冒号“:”，然后跟着协议的名称，如果有多个协议则用逗号“，”分开，如：

```
struct SomeStructure: FirstProtocol,  AnotherProtocol{
    // 结构体的内容
```

```
}
class SomeClass: SomeSuperclass,FirstProtocol, AnotherProtocol{
        // 类的内容
}
```

20.2　对属性的规定

协议可以规定任何“实现类型”必须实现协议里定义的同一类型和名称的实例属性或类型属性。我们在协议中不用指定属性是存储属性还是计算属性，我们只要指定属性的名称和类型，并还需要指定属性是只读属性“gettable”还是可读写属性“gettable and settable”。

- 协议规定：可读写属性“gettable and settable”，“实现类型”不能是常量存储属性或只读计算属性。
- 协议规定：只读属性“gettable”，“实现类型”可以是任何类型的属性，并且我们为属性增加写入属性“settable”也是可以的。

协议里添加属性，必须使用“var”关键字，然后是属性的名称和类型。如果指定的是可读写属性“gettable and settable”，那么使用“{ get set }”来表示，如果是只读属性“gettable”，那么使用“ { get } ”来表示，如：

```
protocol 一些协议{
    var 可读写属性: Int {get set}
    var 只读属性: Int {get}
}
```

这里，我们定义一个名为“一些协议”的协议，里面定义了两个属性，分别是“可读写属性”和“只读属性”，当然，我们除了这些实例属性，还可以给协议添加“静态属性”，如：

```
protocol 另外的协议{
    static var 一些类型属性: Int {get set}
}
```

这里，我们在“另外的协议”里添加了一个静态属性“一些类型属性”，只要在属性最前面添加关键字“static”就可以了。

下面是一个规定了实例属性要求的协议：

```
protocol 全名{
    var 姓名: String {get}
}
```

在上述代码中，我们定义了一个名为“全名”的协议，里面定义了“姓名”这个字符串类型的只读属性。

下面，我们创建一个结构体来实现这个协议，如下：

```
struct 人: 全名 {
```

```
    var 姓名: String
}
let 欧阳风 = 人(姓名: "欧阳风")
print("这个人的姓名: \(欧阳风.姓名)")
```

输出：

这个人的姓名：欧阳风

在上述例子中，我们创建“人”结构体去实现“全名”协议，然后我们实现里面的“姓名”属性，再创建“人”的实例“欧阳风”，在指定构造函数中为属性“姓名”赋予值“欧阳风”，最后我们打印出这个人的姓名。

下面，我们再来看下类是如何实现协议的属性的，如：

```
class 演员: 全名 {
    var 姓: String?
    var 名: String

    init(姓: String? = nil, 名: String){
        self.姓 = 姓
        self.名 = 名
    }

    var 姓名: String{
        return (姓 != nil ? 姓! : "") + 名
    }
}
var 上官无我 = 演员(姓: "上官", 名: "无我")
print("这个演员的姓名: \(上官无我.姓名)")
```

输出：

这个演员的姓名：上官无我

在上述例子中，我们定义了“演员”这个类，也实现了“全名”这个协议，在类中，我们先是定义了两个属性，其中“姓”是个可选类型，然后定义了一个指定构造函数去初始化这两个属性，在实现协议的属性中，当“姓”非空就返回“姓”，否则返回“”和“名”一起组合的字符串。最后，我们创建类“演员”的实例“上官无我”，调用了我们创建的指定构造函数，输入了参数“上官”和“无我”，调用实例的属性“上官无我.姓名”可以看到相关“姓名”属性的输出结果。

20.3 对方法的规定

协议可以规定“实现类型”去实现特定的实例方法和静态方法。协议里定义的方法和正常的方法命名及定义基本一样，不一样的是没有一对花括号“｛｝”和里面的方法体，不能指定参数的默认值。

下面，我们创建一个“静态方法协议”，里面的方法通过在最前面添加关键字“static”定义了协议的静态方法，如：

```
protocol 静态方法协议{
    static func 一些静态方法()
}
```

下面，我们创建一个协议的实例方法，并且该方法返回一个浮点类型的值，如：

```
protocol 随机协议{
    func 随机方法() -> Double
}
```

在上述代码中，我们创建了“随机协议”，里面定义了一个具有“Double”类型返回值的“随机方法”。

下面，我们通过一个类来实现“随机协议”，并且必须实现里面定义的“随机方法”，如：

```
class 随机类: 随机协议 {
    var 随机数 = Double(arc4random())
    func 随机方法() -> Double{
        return 随机数
    }
}
var 随机 = 随机类()
print("随机数是: \(随机.随机数)")
输出：
随机数是：75344827.0
```

在上述代码中，我们使用了类“随机类”来实现协议“随机协议”，在类中，我们定义了一个属性“随机数”，专门获取浮点类型的随机数，然后实现协议里的“随机方法”，直接返回了“随机数”。

20.4　对变异方法的规定

当我们使用结构体或枚举时，不得不在方法中更改所属属性的值，我们就要使用变异方法，在方法定义关键字“func”前添加“mutating”关键字。当然，我们的协议里如果需要变异方法，也要在“func”前添加“mutating”关键字，如下：

```
protocol 开关协议{
    mutating func 转换开关()
}
```

在上述代码中，我们定义了协议“开关协议”，里面定义了一个变异方法“转换开关”。

下面，我们增加一个枚举来实现该协议，如下：

```
enum 开关枚举: 开关协议{
```

```
    case 开,关
    mutating func 转换开关() {
        switch self{
            case .开: self = .关; print("转换后门是：\(self)")
            case .关: self = .开; print("转换后门是：\(self)")

        }
    }
}
var 门 = 开关枚举.关
print("当前的门是：\(门)")
门.转换开关()
```

输出：

当前的门是：开关枚举.关
转换后门是：开关枚举.开

在上述例子中，我们创建了一个枚举“开关枚举”去实现“开关协议”，里面有“开”和“关”两个枚举值，然后实现协议里的变异方法“转换开关”，当自身的值“self”是“开”时，把自身的值“self”改为“关”，当自身的值“self”是“关”时，把自身的值“self”改为“开”。

20.5 对构造函数的规定

协议可以规定“实现类型”去实现特定的构造函数。协议里定义的构造函数和正常的构造函数的命名和定义基本一样，不一样的是没有一对花括号“{}”和里面的方法体，如：

```
protocol SomeProtocol1 {
    init(someParameter: Int)
}
```

在类中，我们实现协议的构造函数，必须添加关键字“required”，如：

```
class SomeClass1: SomeProtocol1 {
    required init(someParameter: Int) {
        //实现协议的构造函数的代码
    }
}
```

如果我们的类既继承了父类，又实现了接口，并且父类的构造函数和接口的构造函数都是一样的，那么我们如何编写代码呢？很简单，先实现接口（用required），再重写方法（用override），如：

```
protocol SomeProtocol2 {
    init()
```

```
    }

    class SomeSuperClass2 {
        init() {
            //父类的初始化方法
        }
    }
    class SomeSubClass2: SomeSuperClass2, SomeProtocol2 {
        // required 用来对协议 SomeProtocol2 的实现，override 用来重写父类
SomeSuperClass2
        required override init() {
            //实现协议和重写父类的构造函数的代码
        }
    }
```

20.6　协议也是一种类型

虽然协议实际上没有实现任何功能，但是任何协议都可以成为我们编写的代码时，作为一种完全成熟的类型来使用。

协议作为一种类型，可以在以下场景中使用协议。

- 作为函数、方法或构造函数的参数类型或返回类型来使用。
- 作为常量、变量或属性的类型来使用。
- 作为数组、字典或其他容器里元素的类型来使用。

下面，我们创建一个具有协议类型的常量，如：

```
    class 浮点数随机{
        let 附加值: Double
        var 随机数: 随机协议

        init(附加值:Double, 随机数: 随机协议){
            self.附加值 = 附加值
            self.随机数 = 随机数
        }

        func 获取随机数()-> Double{
            return 随机数.随机方法() + 附加值
        }
    }
```

在上述例子中，我们创建了一个类“浮点数随机”，里面定义了两个常量，分别是浮点类型常量“附加值”和协议类型常量“随机数”，然后定义了一个指定构造函数“init”，

用来初始化这两个属性。我们创建了一个方法“获取随机数”，用于调用协议里调用的方法“随机数.随机方法()”的值和附加值相加，在这个类中，我们只是利用了协议和协议里定义的方法，并没有具体实现协议和协议方法的具体内容。

下面，我们创建这个类“浮点数随机”的实例，调用里面的方法，获取随机数，如：

```
var 数据 = 浮点数随机(附加值：10, 随机数：随机类())
print("浮点随机数是：\(数据.获取随机数())")
输出：
浮点随机数是：3370761061.0
```

在上述例子中，我们通过上面定义的指定构造函数，添加参数“10”和“随机类()”实例来创建“浮点数随机”的实例“数据”，其中参数“随机类()”是实现协议“随机协议”的类的实例，对于我们在指定构造函数中的参数类型“随机协议”来说，这是一个协议的实现类，当然可以作为参数来使用了。然后，我们创建“数据”实例，通过“数据.获取随机数()”方法就可以获得相关的值了。

20.7 代理

代理：也称“委托”，将一个类或结构体的某些功能委托给另一个类或结构体去实现的一种设计模式。这种设计模式的原理：通过定义一个协议去封装某些功能，让“实现类型”拥有这些被委托的功能。“被委托方”可以响应“委托方”的特定动作，并且“被委托方”可以从“委托方”这里获取相关数据而不用知道“委托方”数据类型。

下面，我们先定义一个类和两个协议：

```
//1.角色类
class 角色{
    var 名称: String
    var 攻击力: Int
    var 防御力: Int

    init(名称: String, 攻击力: Int, 防御力: Int){
        self.名称 = 名称
        self.攻击力 = 攻击力
        self.防御力 = 防御力
    }

}
//2.游戏协议
protocol 游戏{
    var 游戏角色: 角色{ get }
    func 玩游戏()
}
```

```
//3.动作游戏代理协议
protocol 动作游戏代理{
    func 开始游戏(动作游戏: 游戏) -> Int
    func 打怪升级(动作游戏: 游戏, 游戏中使用道具 经验加倍: Bool) -> Int
    func 结束游戏(动作游戏: 游戏) -> Int
}
```

在上述代码中，我们创建了一个“角色”类，里面有属性“名称”、“攻击力”、“防御力”三个属性和一个指定构造函数用来初始化属性值。两个协议分别是“游戏”和“动作游戏代理”，在“游戏”协议里，定义了一个“角色”类型的只读属性“游戏角色”和一个方法“玩游戏”；在“动作游戏协议”里，我们使用了协议“游戏”作为参数的类型在三个不同的方法里。

下面，我们先创建实现“动作游戏代理”协议的类，如：

```
//4.实现动作游戏协议
class 代理练级: 动作游戏代理 {
    func 开始游戏(动作游戏 网络游戏: 游戏) -> Int {
        print("\(网络游戏.游戏角色.名称) 开始游戏，就获得经验 1000")
        return 1000
    }

    func 打怪升级(动作游戏 网络游戏: 游戏, 游戏中使用道具 经验加倍: Bool) -> Int {
        if(经验加倍){
            print("\(网络游戏.游戏角色.名称) 在水帘洞推倒大圣，开启经验加倍，获得经验 5000")
            return 5000
        }else{
            print("\(网络游戏.游戏角色.名称) 在水帘洞推倒大圣，屌丝玩家，只能获得经验 2500")
            return 2500
        }
    }

    func 结束游戏(动作游戏 网络游戏: 游戏) -> Int {
        print("\(网络游戏.游戏角色.名称) 结束打怪，抽奖获得经验 500")
        return 500
    }

}
```

在上述代码中，我们用“代理练级”类来实现“动作游戏代理”协议。在实现“开始游戏”的方法中，通过协议类型参数“网络游戏”来获取原本“游戏”协议类定义的类参数“游戏角色”的属性“名称”，如“网络游戏.游戏角色.名称”来实现，在不知道协议的

“具体实现”的情况下，我们访问了该协议的属性，下面的“打怪升级”和“结束游戏”也一样，获取了相关属性作为输出。在“打怪升级”方法中，还通过参数是否“经验加倍”来判断输出的相关信息和返回经验值的不同。

下面，我们再创建一个类去实现“游戏”协议：

```
//5.实现游戏协议
class 动作游戏: 游戏 {
    // 实现协议的属性
    let 游戏角色 = 角色(名称: "无心法师", 攻击力: 300, 防御力: 100)

    // 定义一个协议类型，代理游戏
    var 代理游戏: 动作游戏代理?
    var 经验值 = 0

    // 实现协议的方法
    func 玩游戏(){
        经验值 += 代理游戏!.开始游戏(动作游戏: self)
        经验值 += 代理游戏!.打怪升级(动作游戏: self, 游戏中使用道具: true)
        经验值 += 代理游戏!.结束游戏(动作游戏: self)
        print("代练完毕，一共获得经验\(经验值)")
    }

}
```

在上述代码中，我们创建了“动作游戏”类来实现“游戏”协议，通过定义常量“游戏角色”来保存协议里的属性“角色”类的实例，再定义一个可选协议类型的参数“代理游戏”，还定义一个变量“经验值”来计算玩游戏后获取的总经验值。我们实现了“游戏”协议的方法“玩游戏”，然后在方法里面，我们调用定义的属性“代理游戏”来调用相关协议里定义的三个方法，在“开始方法”、“打怪升级”、“结束游戏”里，把自身“self”作为参数传入，并在“打怪升级”方法里，游戏是高富帅的人民币玩家，花钱买道具成为可能！直接把“游戏中使用道具”设置为“true”来获取双倍经验值。

下面，我们通过创建相关实例来说明：

```
//6.创建实例调用
let 代练小陈 = 代理练级()
let 太极熊猫 = 动作游戏()
太极熊猫.代理游戏 = 代练小陈
太极熊猫.玩游戏()
输出：
无心法师 开始游戏，就获得经验 1000
无心法师 在水帘洞推倒大圣，开启经验加倍，获得经验 5000
无心法师 结束打怪，抽奖获得经验 500
代练完毕，一共获得经验 6500
```

在上述代码中，我们创建了“代理练级”类的实例“代练小陈”和“动作游戏”类的实例“太极熊猫”，然后调用“太极熊猫”的属性“代理游戏”等于“代理小陈”来实现委托，简单来说，就是把“太极熊猫”这个游戏委托给“代理小陈”来玩。最后调用“太极熊猫”的“玩游戏”方法，可以看到“代理小陈”尽职地把代练的工作做完，完成代练。

20.8　在扩展中添加协议

即使我们没有权限修改或访问已存在的代码，但还是可以在扩展存在的类型中添加相关协议的实现。扩展，可以添加新的属性、方法、下标到已存在的类型中，并可以实现协议所定义的功能。

我们先创建一个协议，如：

```
protocol 结果展示{
    func 显示名称() -> String
}
```

在上述代码中，我们创建了一个“结果展示”协议，里面定义了一个返回字符串类型的方法“显示名称”。

然后，我们扩展之前定义好的类“动作游戏”，并实现“结果显示”这个协议，如：

```
extension 动作游戏: 结果展示{
    func 显示名称() -> String {
        return 游戏角色.名称
    }
}

let 扩展动作游戏 = 动作游戏()
print(扩展动作游戏.显示名称())
输出：
无心法师
```

在上述代码中，我们实现的“显示名称”方法里，调用了“动作游戏”类里的属性“游戏角色.名称”用于返回结果，然后创建了这个类的实例“扩展动作游戏”，通过调用实现协议的方法“显示名称”，把游戏角色的名称显示出来。

20.9　协议类型的集合

协议，可以作为一种数据类型存储在各种集合中，如数组或字典，表示结合中的元素类型和协议类型。

我们继续添加扩展协议“结果展示”，如下：

```
extension 代理练级: 结果展示{
```

```
    func 显示名称() -> String {
        return "代理练级"
    }
}
extension 角色: 结果展示{
    func 显示名称() -> String{
        return self.名称
    }
}
let 扩展代理练级 = 代理练级()
let 扩展角色 = 角色(名称: "周公", 攻击力: 400, 防御力: 300)
```

在上述代码中，我们再为“代理练级”和“角色”类扩展协议“结果展示”，都在相关代码中实现了“显示名称”的方法，然后创建了相关实例“扩展代理练级”和“扩展角色”。

下面，我们创建一个协议“结果展示”的数组，并把之前创建的“扩展动作游戏”、“扩展代理练级”和“扩展角色”放到数组里，然后循环调用实例里的“显示名称”方法，打印结果，如：

```
let 名称集合: [结果展示] = [扩展动作游戏,扩展代理练级,扩展角色]
for 名称 in 名称集合{
    print(名称.显示名称())
}
输出：
无心法师
代理练级
周公
```

20.10 协议的继承

协议可以继承一个或多个其他类型的协议，语法与类的继承相似，继承多个协议之间我们用逗号“，”分隔，如：

```
protocol InheritingProtocol: SomeProtocol1, AnotherProtocol1 {
    // 实现协议的代码
}
```

下面，我们再创建一个新的协议来继承“结果展示”协议，如：

```
protocol 显示内容: 结果展示{
    func 获取经验() ->String
}

extension 动作游戏: 显示内容{
```

```
    func 获取经验() -> String {
        let 输出 = 显示名称() + "获取了 \(self.经验值) 经验值"
        return 输出
    }
}
let 扩展动作游戏2 = 动作游戏()
扩展动作游戏2.经验值 = 9999
print(扩展动作游戏2.获取经验())
输出：
无心法师获取了 9999 经验值
```

在上述代码中，我们创建了“显示内容”协议，并继承自“结果展示”协议，里面定义了一个返回字符串类型的方法“获取经验”。然后我们扩展类“动作游戏”实现我们新创建的“显示内容”协议，由于我们之前扩展类“动作游戏”已实现了“结果展示”协议里的“显示名称”方法，所以这里不用再次实现，只要实现我们“显示内容”里的“获取经验”即可。我们在实现“获取经验”方法中，调用了之前扩展实现的“显示内容”和自身“self”的属性“经验值”一起组合起来的字符串。通过创建“动作游戏”的实例“扩展动作游戏 2”，先给经验值属性赋予“9999”，然后再打印“获取经验”的信息。

20.11　协议类型的组合

如上所述，我们知道了协议是一种类型，有时我们可能需要这种协议类型必须实现一个以上的协议，这样，我们称之为协议类型的组合，协议类型的组合形式使用一对尖括号“<>”，里面多个协议之间用逗号“，”分开，如“<someprotocol，anotherprotocol>”。

下面，我们定义两个协议、一个结构体和一个方法，看看我们是如何使用协议的组合的，如：

```
protocol 姓名协议{
    var 姓名: String {get}
}
protocol 年龄协议{
    var 年龄: Int {get}
}
struct 人员: 姓名协议,年龄协议 {
    var 姓名: String
    var 年龄: Int
}
func 生日(相关人员: 姓名协议 & 年龄协议){
    print("\(相关人员.姓名) 祝您 \(相关人员.年龄) 岁生日快乐！")
}
```

```
let 生日人员 = 人员(姓名: "郭碧停", 年龄: 24)
生日(相关人员: 生日人员)
输出：
郭碧停 祝您 24 岁生日快乐！
```

在上述代码中，我们创建了两个协议：“姓名协议”里面定义了只读字符串属性“姓名”；“年龄协议”里定义了只读整型属性“年龄”。定义了结构体“人员”实现了上面两个协议，并实现了“姓名”和“年龄”属性。我们创建了一个函数“生日”，函数的参数为协议类型的组合“<姓名协议，年龄协议>”，然后输出相关信息。我们先创建结构体的实例，在构造函数中输入相关人员的姓名和年龄，然后通过调用“生日”这个带有协议类型组合的参数，把结构实例“生日人员”作为参数传入即可输出相关信息，因为结构体本身就实现了该方法的协议类型的组合。

20.12 检查协议的一致性

我们可以使用“is”操作符检查协议和相关“实现类型”是否一致，使用“as”操作符对协议进行转型操作，但是，我们必须遵循以下几点要求。

- “is”操作符检查协议和“实现类型”是否一致，一致时返回“true”，不一致时返回 false。
- “as?”操作符代表协议是可选类型的值，当实现类型没有实现协议时返回 nil。
- “as!”操作符代表我们确定了实现类型实现了协议，我们强制转换类型。

下面，我们通过一个例子来说明，如：

```
protocol 下厨{
    var 做粤菜: String {get}
}

class 白领:下厨 {
    var 姓名: String
    init(姓名: String){
        self.姓名 = 姓名
    }
    var 做粤菜: String { return "\(姓名) 在做：扬州炒饭" }
}
class 蓝领: 下厨 {
    var 姓名: String
    init(姓名: String){
        self.姓名 = 姓名
    }

    var 做粤菜: String { return "\(姓名) 在做：清蒸鲈鱼" }
```

```
}
class IT人员 {
    var 姓名: String
    init(姓名: String){
        self.姓名 = 姓名
    }

    var 做粤菜: String { return "\(姓名) 外卖：尖椒炒肉" }
}
```

在上述代码中，我们创建了一个协议“下厨”，里面定义了一个只读类型的字符串属性“做粤菜”，然后分别定义了三个类“白领”、“蓝领”和“IT人员”。三个类里都定义了字符串类型的属性“姓名”和一样带“姓名”的指定构造函数，都是初始化我们定义的属性“姓名”，然后都有一个和协议定义一样的字符串属性“做粤菜”，并都返回了相关信息。三个类不同的是，“白领”和“蓝领”这两个类实现了协议“下厨”，而“IT人员”类没有实现该协议。

然后我们创建这三个类的实例，放到数组中，再迭代出来，如：

```
let 写字楼人员: [AnyObject] = [白领(姓名: "小波"),蓝领(姓名: "华仔"),IT 人员(姓名: "希达")]
for 人员 in 写字楼人员{
    if let 做菜人员 = 人员 as? 下厨{
        print(做菜人员.做粤菜)
    }else{
        print("IT 人员只能叫外卖了！")
    }
}
```

输出：

```
小波 在做：扬州炒饭
华仔 在做：清蒸鲈鱼
IT 人员只能叫外卖了！
```

在上述代码中，我们迭代数组“写字楼人员”时，获取了数组里相关的类“人员”。然后我们通过操作符“as?”，对“人员”的可选类型“下厨”协议进行判断，如果不返回nil，则调用“做粤菜”来输出相关信息，如果返回nil，则证明了该类没有实现“下厨”这个协议。就算类里面的方法和属性都和协议的定义一样，只要没有实现协议的定义，就一概为nil，这里，我们只有类“IT人员”没有实现“下厨”这个协议，所以打印出“IT人员只能叫外卖了！”这个信息。



第 2 篇



iOS 11 基础开发篇

第 21 章
入门级

21.1 开发环境配置

工欲善其事必先利其器，弄斧就得去班门，全套装备齐全才好作战。

不幸的是，iOS 开发需要苹果专用的 MacOS 系统，所以需要苹果官方出售的计算机。官方有：Mac Mini（自配显示器）、MacBook Air、MacBook、MacBook Pro、iMac、Mac 5K、Mac Pro。苹果计算机、手机体验虽好，但作为一个“屌丝”程序员，还是感叹价格比较贵，有没有“山寨”的？幸运的是，无所不能的黑客把 MacOS 系统进行了修改，让它可以安装到普通的 PC 上，俗称黑苹果。可以在现有的 PC 上安装 MacOS 系统，但并非所有硬件都能兼容。淘宝上有这样的服务，在 200 元左右。或者新购 PC，因为 MacOS 系统字体设计的原因，普通显示器显示有颗粒感，4K 显示器才能完美显示，加起来 1 万元以内也能搞定，性价比很高，但其缺点是系统一旦更新，还要求助高手帮忙升级，否则就会黑屏。

苹果平台的开发自成一格，XCode 是苹果官方推出的开发工具，在 MacOS 自带的 Mac App Store 里可以搜索 XCode，即可下载安装。XCode 安装包很大，下载及安装都较慢，读者要有耐心。安装过程是自动的，安装完毕后启动即可。

XCode 包含 iOS SDK、源码编辑器、UI 设计器、调试器、性能分析器、iPhone/iPad 模拟器、打包和上架工具、开发文档等，非常完美，XCode 图标如图 21-1 所示。



图 21-1

除此之外，苹果另外一个套路就是账号系统要求，我们需要注册一个苹果开发者账号，完全免费，网址是 http://developer.apple.com/。苹果官方开发者界面如图 21-2 所示。

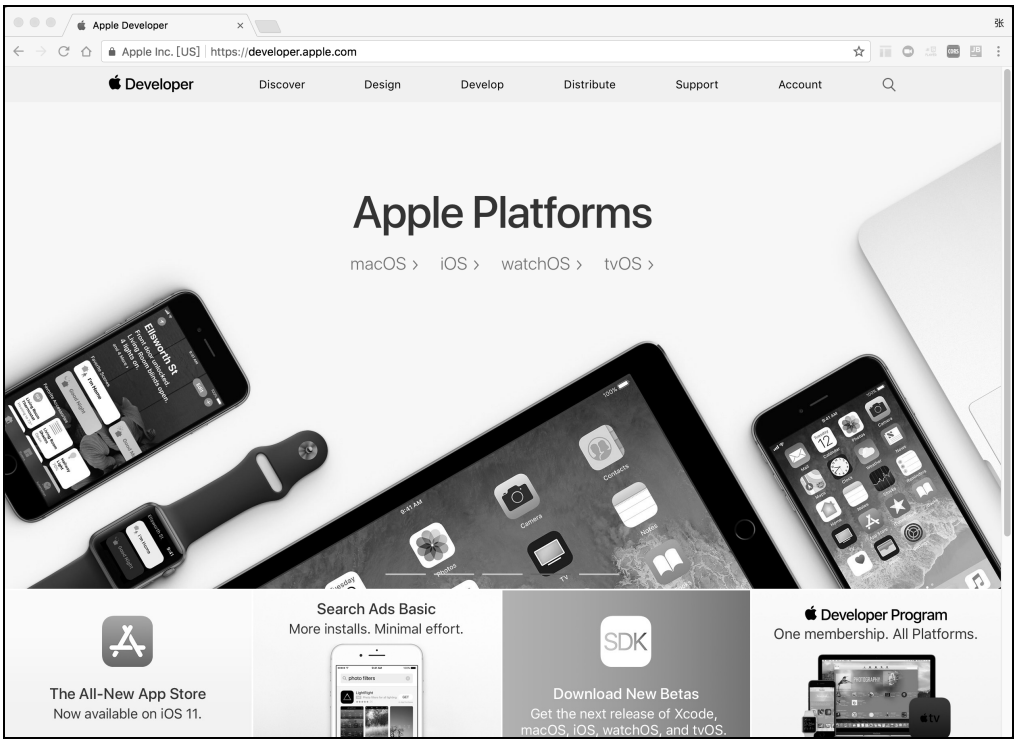


图 21-2

单击右上角的“Account”链接，即可进入账号注册页面，单击“CreateAppleID”选项，根据提示完成账号注册。苹果开发者账号主要用于下载开发证书、下载 Beta 版本 iOS 和 Xcode、真机测试。另一个用途就是账号的增值服务，年费是 688 元人民币，主要用于上传 App，以及配置推送证书、启用各种特色功能（Apple Pay、App 组、Passbook 等）。

基本上整个环境配置完成下来，感觉自己被迫成了一个人民币玩家。

21.2 Playground

在 2014 年 6 月前，iOS 开发只能用比较传统的 Objective-C 语言，语法相对过时、烦琐。Swift 结合了 JavaScript、Python、C#等优秀编程语言的优点，类型安全、泛型、可选类型、闭包等现代编程语言必备的特性，以及自身优秀的面向协议编程特色，同时保持了语法的简洁明快，这也就是为什么大家这么喜欢 Swift 的直接原因。

1. Objective-C 语法例子

```
NSString *weatherCondition = @"今天好冷！";
```

```
        NSString *drinking = @"咱买奶茶喝吧？";

        const int count = 5;
        int price = 10;

        int total = (count * price);

        NSString *suggest = [NSString stringWithFormat:@"%@%@一共%d块",
weatherCondition , drinking, total];

        NSLog(@"%@", suggest);
```

2. Swift 语法对比

```
let 天气 = "今天好冷"
let 饮品 = "咱买奶茶喝吧?"

let 数目 = 5
let 价格 = 10

let 建议 = 天气状况+饮料 + "一共\(价格 * 杯数)块"

print(建议)
```

Playground 是苹果专门为了方便学习 Swift 而开发的交互式编程环境，输入完代码立即就可以在右侧看到结果，内嵌在 XCode 中，如图 21-3 所示。

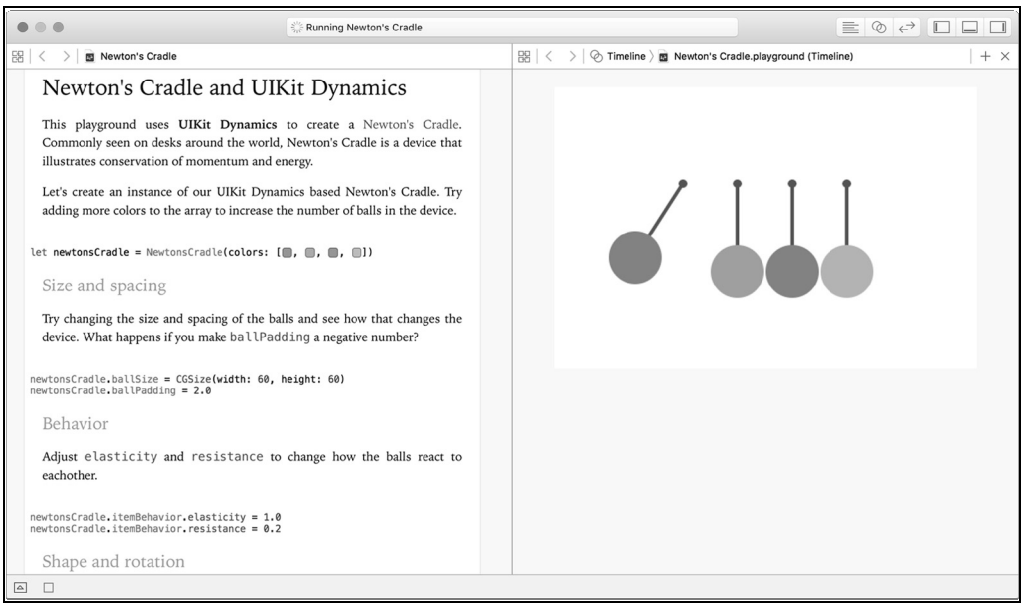


图 21-3

在启动 XCode 后的欢迎界面中，单击第一个选项“Get Started with a playground”，如图 21-4 所示。



图 21-4

选一个 Playground 模板，如图 21-5 所示。

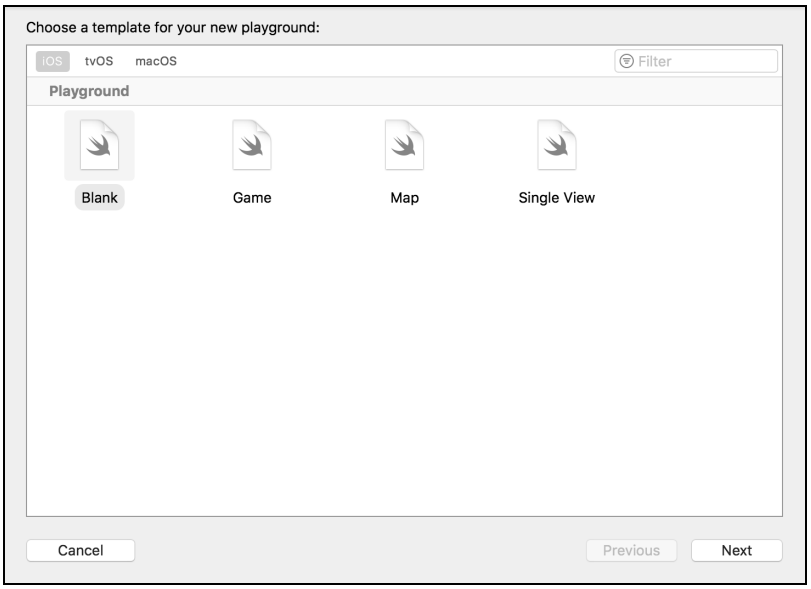


图 21-5

XCode 提供 4 个 Playground 模板，即空模板、游戏模板、地图模板和单视图模板。

选择“Game”，单击“Next”按钮，选一个存储位置（如桌面），单击“Create”按钮，如图 21-6 所示。

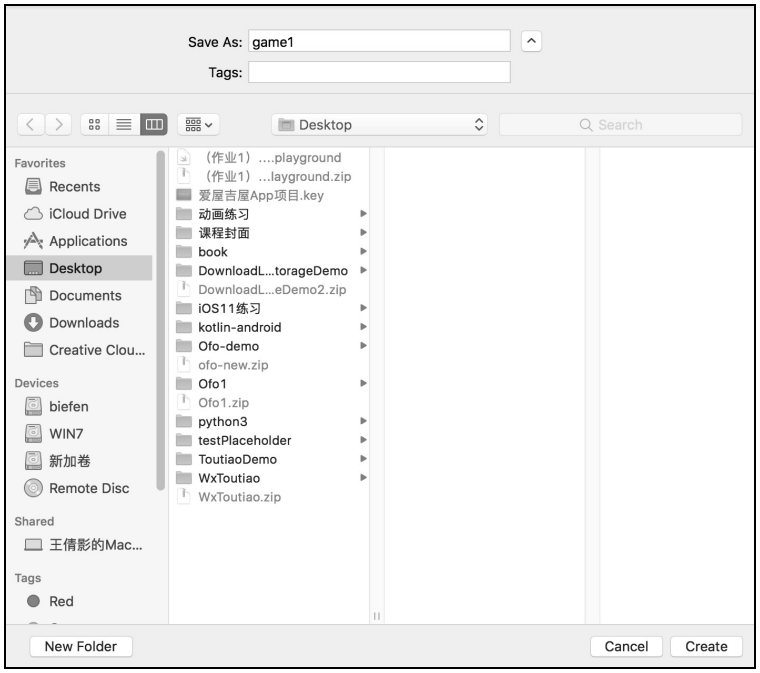


图 21-6

这时可以看到一些代码，单击 XCode 右上角的双环按钮，可以打开代码的 LiveView（实时预览）窗格，如图 21-7 所示。

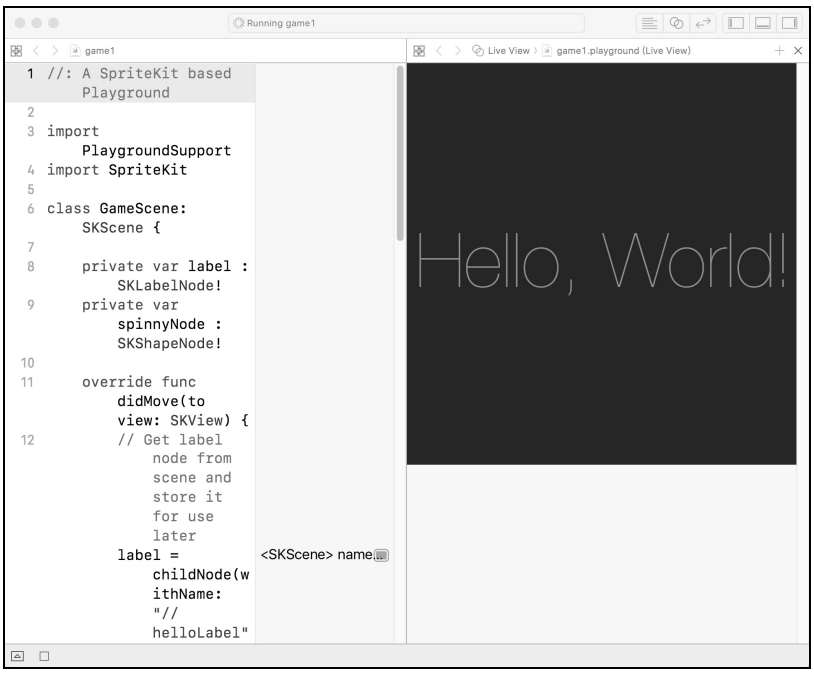


图 21-7

免费获得一个闪烁版的 Hello World，用鼠标在 HelloWorld 上点击滑动，会出现很多彩色的圆角小方块。这一大段代码使用了 iOS 中游戏相关的开发知识。当然现在读者可能看不懂这些代码，这里只是展示了 Playground 的强大之处。

下面我们介绍一个简单的例子，重新选择一个空（Blank）模板来画一个圆角按钮，代码如下。

```
import UIKit

let 矩形 = CGRect(x: 0, y: 0, width: 200, height: 50)
let 订购按钮 = UILabel(frame: 矩形)

订购按钮.text = "立即订购"
订购按钮.textColor = .white
订购按钮.backgroundColor = .purple
订购按钮.textAlignment = .center
订购按钮.layer.cornerRadius = 10
订购按钮.clipsToBounds = true
```

注意，在最后一句代码的最右侧，单击方形图标，即变成蓝色，将“立即订购”按钮的最终样式显示在代码下方。

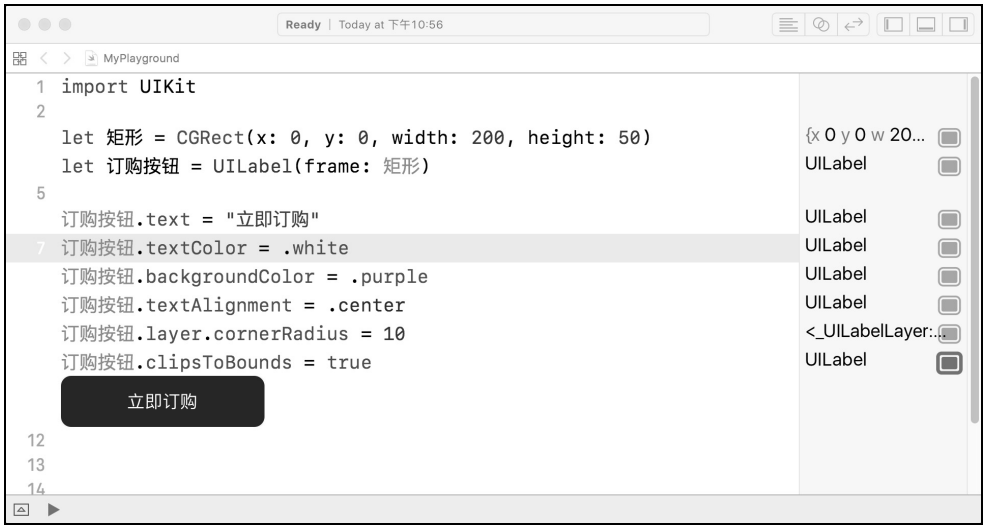


图 21-8

21.3 第一个 App

Playground 虽然好，但我们还是希望马上进入第一个能在 iPhone 上运行的 App，效果如图 21-9 所示。



图 21-9

1. 新建工程

XCode 是以工程为单位来开发 App 的。从 XCode 欢迎界面中，或者执行 XCode 菜单 File→New→Project 都可以新建工程。XCode 的工程模板比 Playground 的模板更丰富，有 Application（单视图 App、游戏 App、AR App 等）和 Framework & Library（框架和库）。

我们从最简单的单视图 App，也就是只有一个页面的 App 开始介绍。

选择工程模板，如图 21-10 所示。

设置新工程的信息，如图 21-11 所示。

（1）Product Name：工程名称，一般是 App 的英文名。这里我们输入 HelloWorld，注意这里避免填写中文。

（2）Team：团队名称，默认是 None。这里可以单击下拉箭头，登录之前注册的苹果开发者账号，就会显示开发团队名称。

（3）Organization Name：组织名称。填写用户所在组织的英文名称或用户的个人工作室的名称。

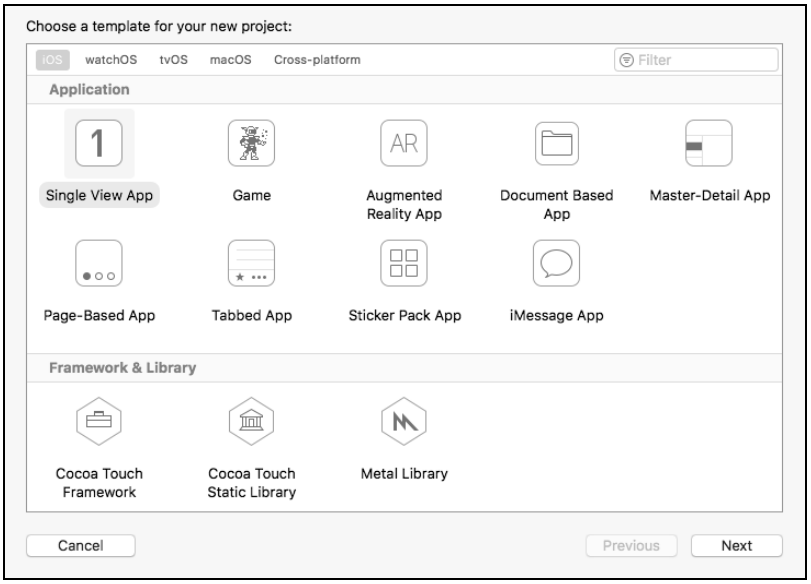


图 21-10

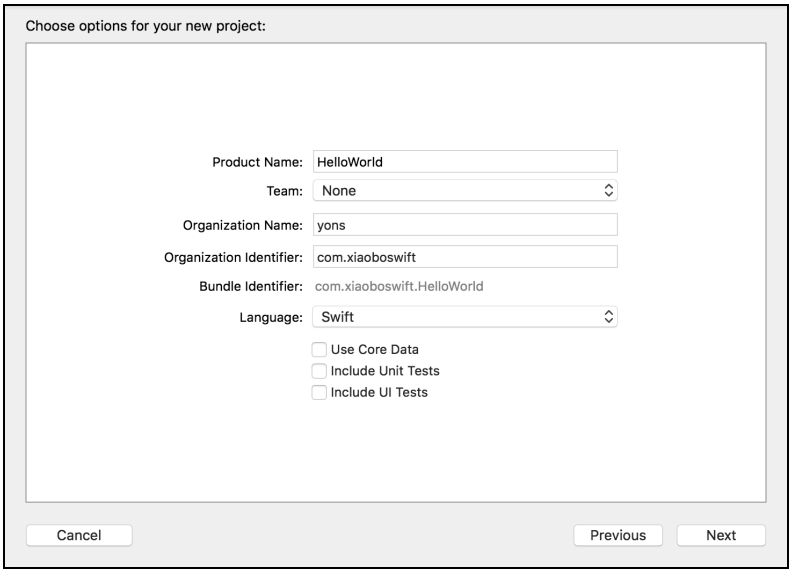


图 21-11

（4）Organization Identifier：组织标识符。组织名称重复率可能很高，但苹果公司要求标识符不能重复（提交 AppStore 上架时），这时一般使用不会重复的国际域名的倒序模式。例如，笔者的个人网站是 xiaoboswift.com，倒序就写成 com.xiaoboswift。注意，下方会自动生成 BundleID（套装 ID），即 com.xiaoboswift.HelloWorld。

（5）Language：默认使用 Swift 来开发。其他选项保持默认状态即可，单击“Next”按钮保存工程。

保存后的 HelloWorld 工程如图 21-12 所示，如果刚才组织 ID 写错了，还可以在这里生

成的 BundleID 中修改。

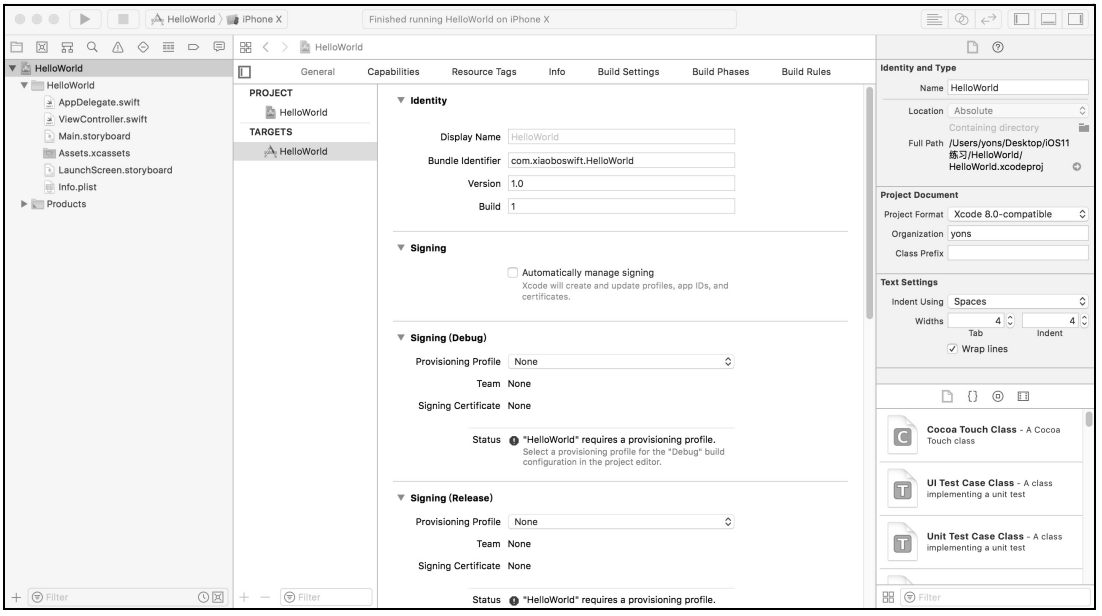


图 21-12

2. 熟悉 XCode

XCode 的功能虽然多，但学会并不困难。我们先来熟悉最常用的几个功能区域，如图 21-13 所示。左侧是工程文件导航窗格，所有工程生成的文件都列在这里。

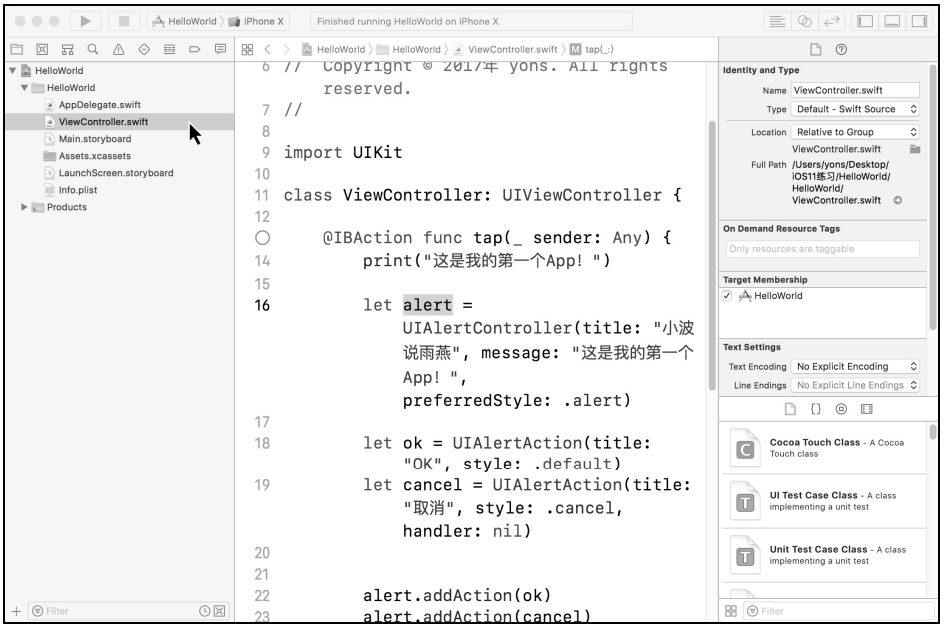


图 21-13

选中.swift 的文件，XCode 在中间区域则显示此文件的 Swift 源代码，这个区域称源码编辑器。可以按快捷键 Commmand 和+/-来缩放源码编辑器中字体的大小。

如图 21-14 所示，选中 Main.storyboard 文件，则中间区域显示为 App 的主场景设计预览，在正下方可以选择预览设备及横盘状态。这个区域称界面构建器（InterfaceBuilder），今后我们在代码中看到以 IB 开头的，都表示与界面相关。

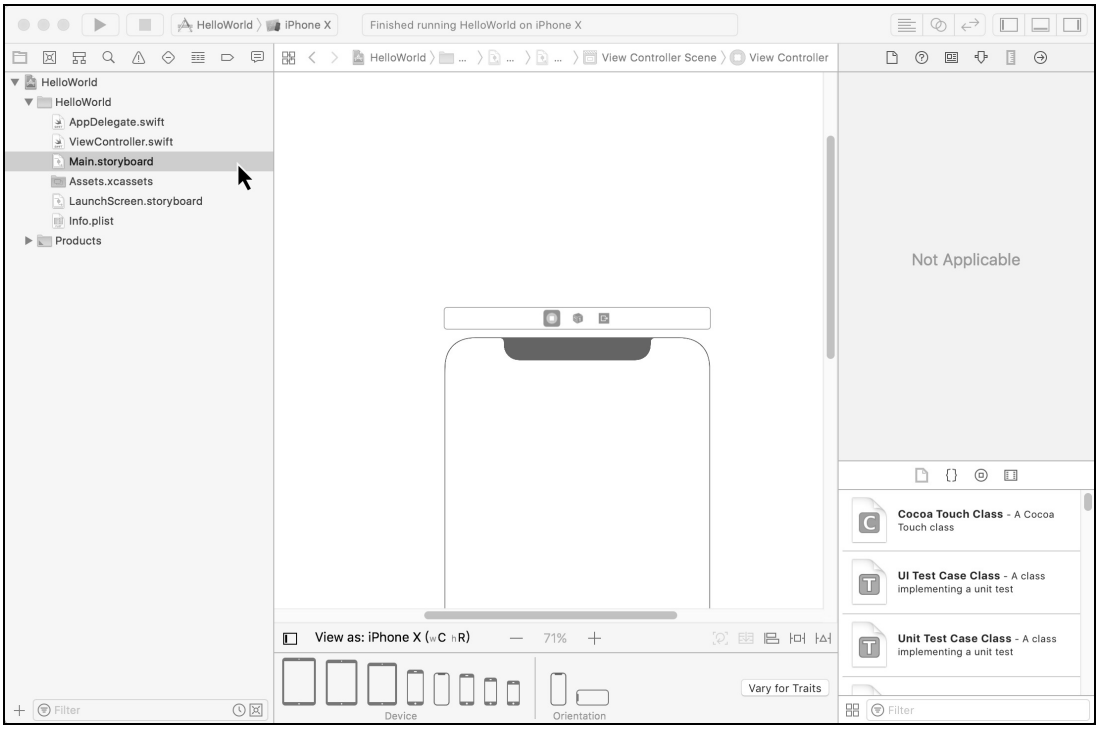


图 21-14

这些功能区域都可以根据需要显示或隐藏，总开关位于 XCode 的右上角，一共有 6 个按钮，如图 21-15 所示。

按钮从左到右分别表示

（1）只显示代码编辑器。

（2）双环图标可以同时显示代码编辑器和界面构建器。

（3）显示代码变更记录。

（4）显示工程文件导航。

（5）运行控制台。

（6）属性栏。

代码和界面设计好了！那么怎么运行呢？在 XCode 的左上角一块区域可以对 App 进行运行控制，包括运行、停止，选择模拟器或真机，如图 21-16 所示。



图 21-15

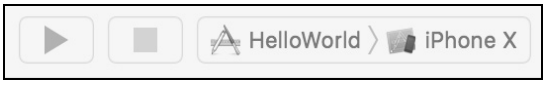


图 21-16

3. 界面设计

一个 App 界面的好坏有时可能决定一切。然而从开发者的角度看，美工送来的漂亮的 UI 效果图，并不是可以瞬间实现的。如何实现，就需要非常熟悉 iOS 中界面设计的一些方法和技巧了。

作为第一个入门 App，界面设计非常简单，只要在页面中部显示一个按钮，按钮文字是“我的第一个 App”，并可以单击。

选择 Main.storyboard 来显示界面构造器，App 的每个页面都称为一个“场景（Scene）”，在设计预览左侧的窗格是“场景结构”，这里显示了场景中各元素的层次。就好像我们去电影院观看芭蕾舞的感觉，把每一个场景串联起来，就成了一个完整的表演。这就是为什么叫 storyboard（剧情板）的原因。现在这里只有一个场景，而且是进入 App 的第一个画面，所以在左侧有一个大箭头，表面它是入口场景，在场景预览窗格左下角有一个小图标，可以显示或隐藏场景结构窗格，如图 21-17 所示。

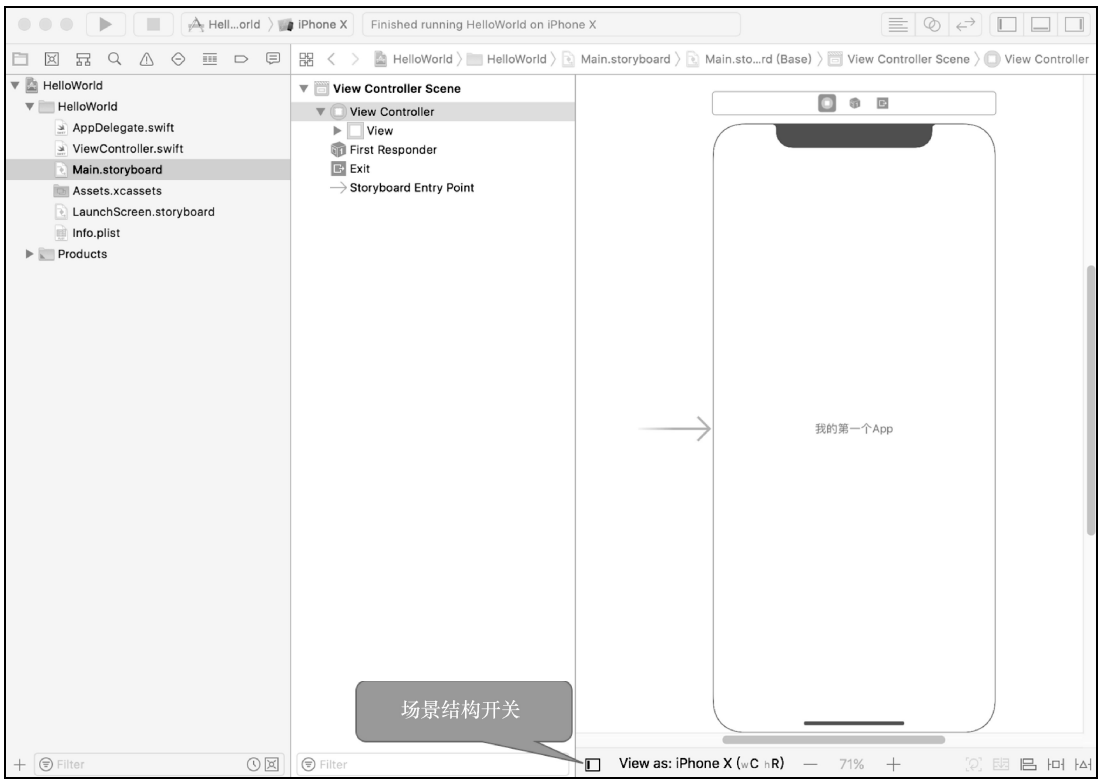


图 21-17

读者应该已经发现了，按钮从哪里来呢？没错，全部可以采用拖拽式设计，与学习 Photoshop 类似，右下角的一块区域中就有一些 UI 组件，找到我们需要的“Button”，把它拖到主场景上，如图 21-18 所示。



图 21-18

拖过来的按钮文字是 Button，双击它即可编辑。或者选中 Button，在右侧的属性栏中找到 Title（标题），把默认的 Button 改为我们需要的文字即可。

要明确一点，拖到场景的 UI 元素的预览位置，并不是实际运行后的位置，需要精准定位后，显示的位置才是实际运行位置。在预览窗格右下栏包含 5 个小图标（也称约束栏），选中 Button 后单击约束栏的第 3 个图标，弹出一个气泡，，标题是“添加新的对齐约束”。勾选“Horizontally in Container”和“Vertically in Container”，再单击“Add 2 Contstraint”按钮，如图 21-19 所示。

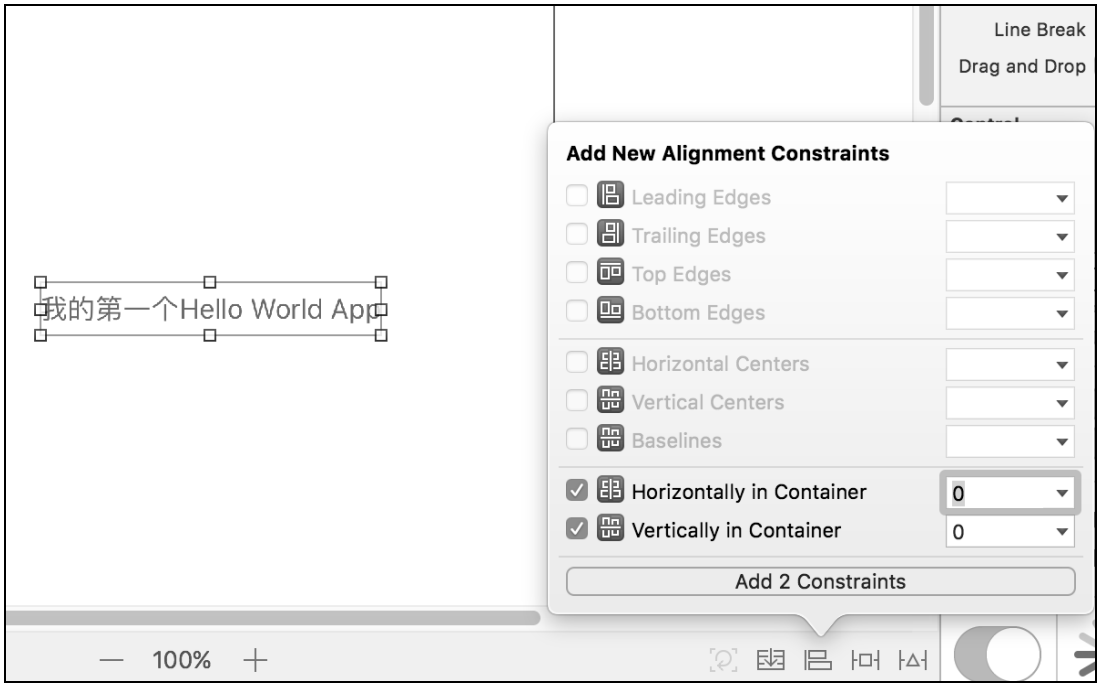


图 21-19

可能有的读者看到这些英文就晕了，其实很简单，这就是指定这个按钮在水平（Horizontally）和垂直（Vertically）居中这两条定位，按钮因为就在场景的最上面，所以它的容器（Container）就是指场景本身。一条定位称为一个约束（Constraint），所以会出现“Add 2 Constraits”按钮。

添加约束后，约束位置与当前位置不一致，会出现橙色提示框，虚线为约束位置，如图 21-20 所示。



图 21-20

单击右下方的第一个按钮即可刷新到约束位置，如图 21-21 所示。



图 21-21

现在选中 Button 会发现有两条蓝色实线贯穿，这表明此元素已有了两个约束。可以在属性栏的尺寸分栏里看到更详细的描述，包括含元素轮廓的约束缩略图、约束列表、约束的数目，如图 21-22 所示。

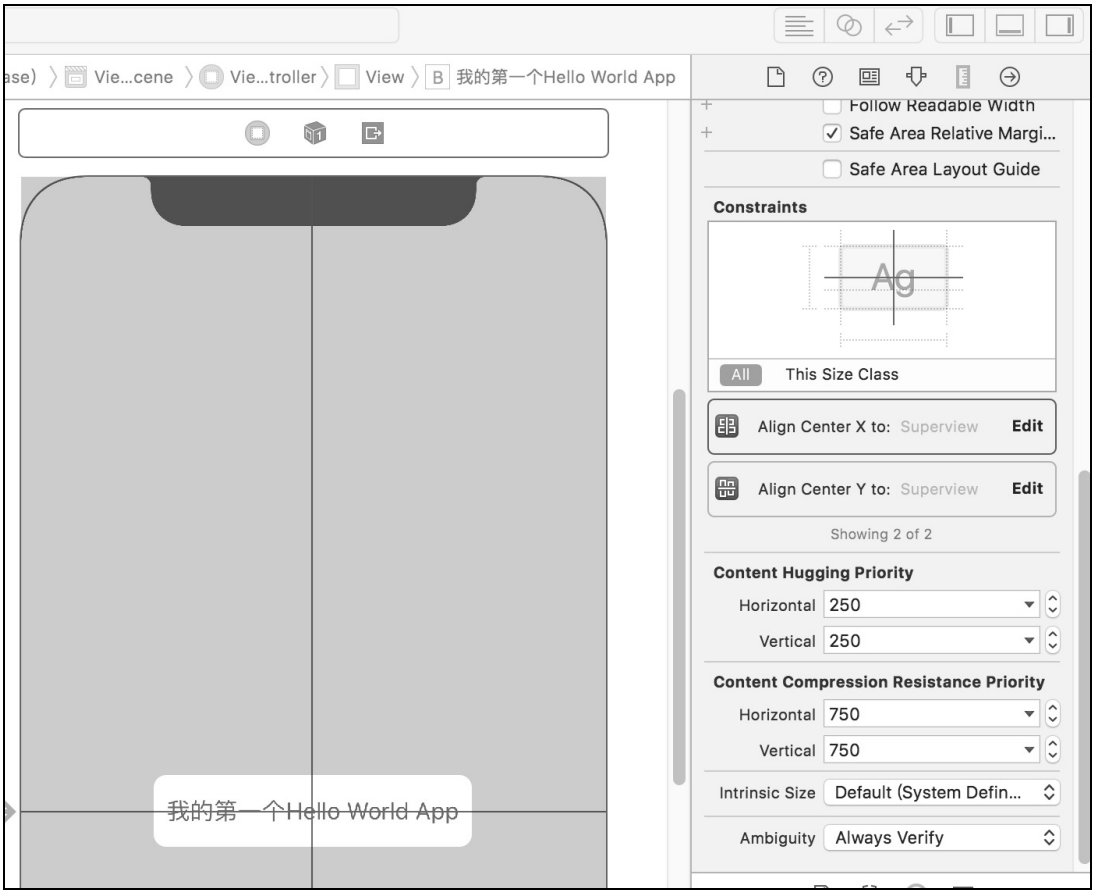


图 21-22

现在运行栏选中 iPhone X 模拟器，看看运行效果，如图 21-23 所示。

讲到这里，我们可以尝试实现如图 21-24 和图 21-25 所示的效果。

要求：在按钮上加入滑块、菊花、开关，依次位于屏幕上半部分四分之一、二分之一、四分之三处，模拟器横屏（Command + 左箭头）时相对位置也不能改变。

其实只要调整该元素的垂直对齐约束中的系数（Multiplier）即可，如 1/4、1/2、3/4，如图 21-26 所示。



图 21-23



图 21-24



图 21-25

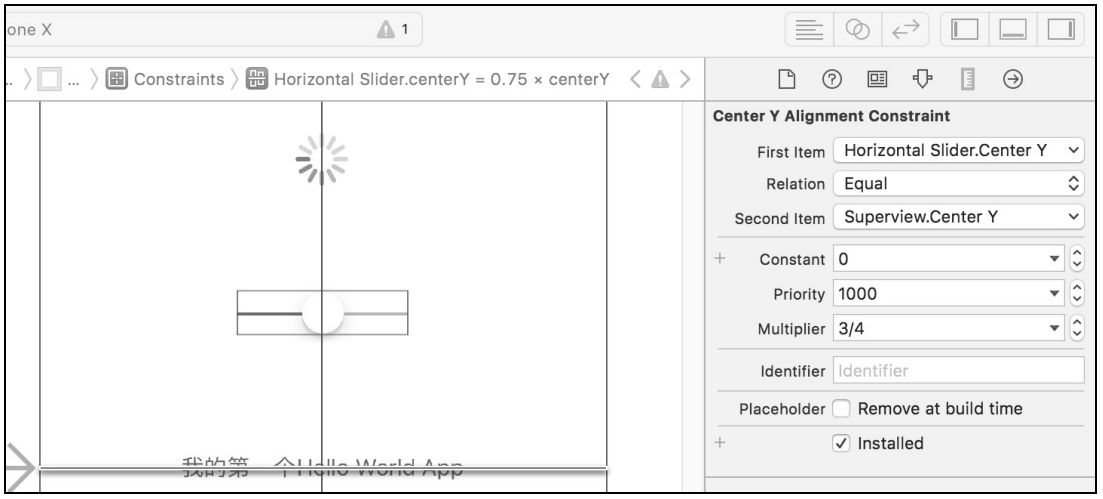


图 21-26

21.4 创建用户交互

第一个 App 上手如此容易，让我们小有成就感，但目前还没有输入任何代码，如何让 App 与用户交互起来呢？

在这之前又要普及一些术语了，我们已经了解了 Main.storyboard 中包含主场景的设计。在场景结构栏中，我们会发现最上层的称 View Controller Scene，如图 21-27 所示。

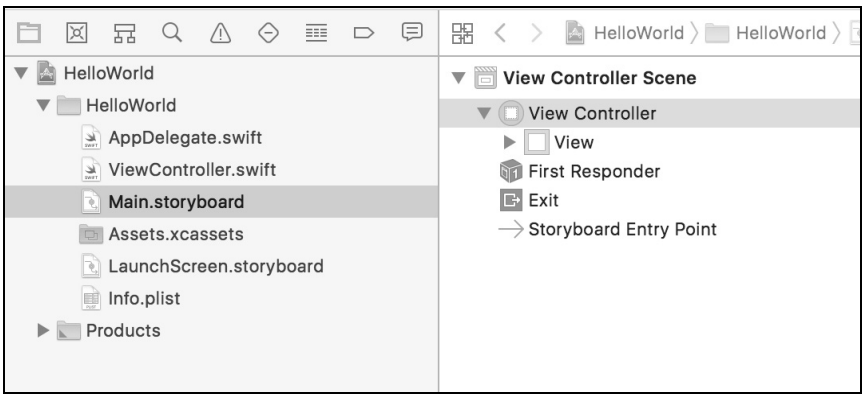


图 21-27

这个 View Controller 是什么意思？

从现在开始，我们涉及术语的转变，在 iOS 中，所有界面（包括 Button 这样的 UI 元素）都被称为 View（视图）。

控制这些 View 的代码称 View Controller（视图控制器）。第一个 App 主场景对应的控

制器代码就是 ViewController.swift，这是项目自动配置好的。

因为第一个 App 场景结构比较简单，下面设计一个稍微复杂一点的场景。

例如，场景上多了一个 Switch（开关），一个 Container View，内含两个 Image View 用于显示两张图片。

场景结构中最上层的 View，我们称为根视图（root view），有且仅有一个。

根据实际应用的需要，可以在根视图中加上诸如 Button、Container View 这样的子视图（child views），子视图可以继续延伸下去，如图 21-28 所示。

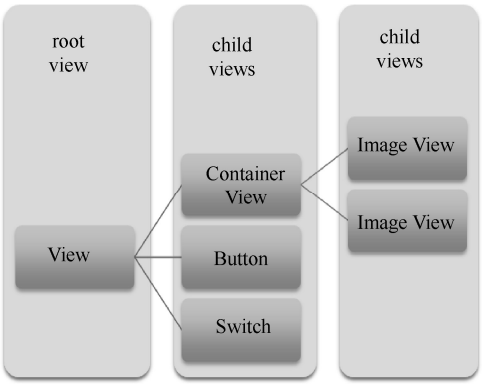


图 21-28

根视图默认由 ViewController.swift 来控制，每个子视图都可以有单独的控制器。

但有时需要把部分控制权交于根视图控制器。

例如，按钮通常需要与用户互动，这就需要连接根控制器，说明该按钮是如何“动作（Action）”的。这个动作是单击还是双击，或者长按、压按等。

我们想要的互动很简单，单击按钮，控制台显示一行字“你点了我”。

下面来看看如何实现。

（1）单击右上角的双环按钮，同时打开界面和代码两个窗格，此时可以把工程文件和属性栏关闭以留出更大的屏幕空间显示。

（2）按住【Ctrl】键拖动 Button 到 ViewController.swift 相邻 class 的下一行，弹出一个小窗口询问连接（Connection）类型，选择 Action。默认动作（Event）是 TouchUpInside，即单击。给这个动作输入一个名称（Name），如 tapMe（点了我）。然后单击“Connect”按钮创建连接。

（3）如果连接成功，则 ViewController.swift 中会有一个方法出现：

```
@IBActionfunc tapMe(_ sender: Any) {
}
```

（4）接着在方法中加入：

```
print("你点了我，晚上吃鸡！")
```

这时可以运行模拟器，单击按钮的结果如图 21-29 所示。



图 21-29

总结：本章从介绍 XCode 开始，到第一个 App 工程的搭建、界面设计、元素定位、场景结构层次及用户交互，让我们有了对 iOS 上 App 开发的初步印象。这是一个很好的开始，我们可以进入下一级了！

第 22 章
初　级

22.1　Auto Layout

前面简单介绍过定位，用到了“对齐约束”，它属于一种相对定位。

本章以在屏幕右下角添加一个按钮为例，介绍 Auto Layout 的其他功能。

在 iPhone 4 以前，屏幕都是固定的尺寸，绝对定位就非常方便。随着屏幕清晰度的提升，只有绝对定位就显得力不从心了。Auto Layout（自动布局）有了更抽象的相对定位，如表 22-1 所示。

表 22-1

iPhone	像素点之比	尺　寸	定　位
4 之前	1:1	3.5	绝　对
4/4s	2:1	3.5	Auto Layout
5/5s		4	
6/7		4.7	
Plus/X	3:1	5.5/5.8	

从组件库中拖曳一个 Label 到场景右下角，如图 22-1 所示。

单击右下方第四个图标，弹出“添加新约束”对话框，在右和下的两个框中都填入 50，再单击“Add 2 Constraints”按钮，如图 22-2 所示。

这两个约束就是指给 Label 增加了到右侧和底边各 50 个点的距离。

在尺寸栏中查看刚刚添加的两个约束，Align Trailing/Bottom to：Safe Area，Equals：50，如图 22-3 所示。Trailing 是指最右侧，Bottom 是指底边，Safe Area（安全区域）是什么呢？

打开场景结构，主场景根视图下的第一个子视图就是 Safe Area（安全区域）。

因为 iPhone X 带有刘海和底部返回，不属于 App 的功能显示范围。为了解决由此带来的 App 适配问题，iOS 11 推出 Safe Area。它是指除去刘海、底部返回，顶部导航栏和底部工具栏的区域，在此区域放置元素是“安全”的。单击 Safe Area，可在各预览设备下查看其范围（以灰底背景显示），如图 22-4 所示。

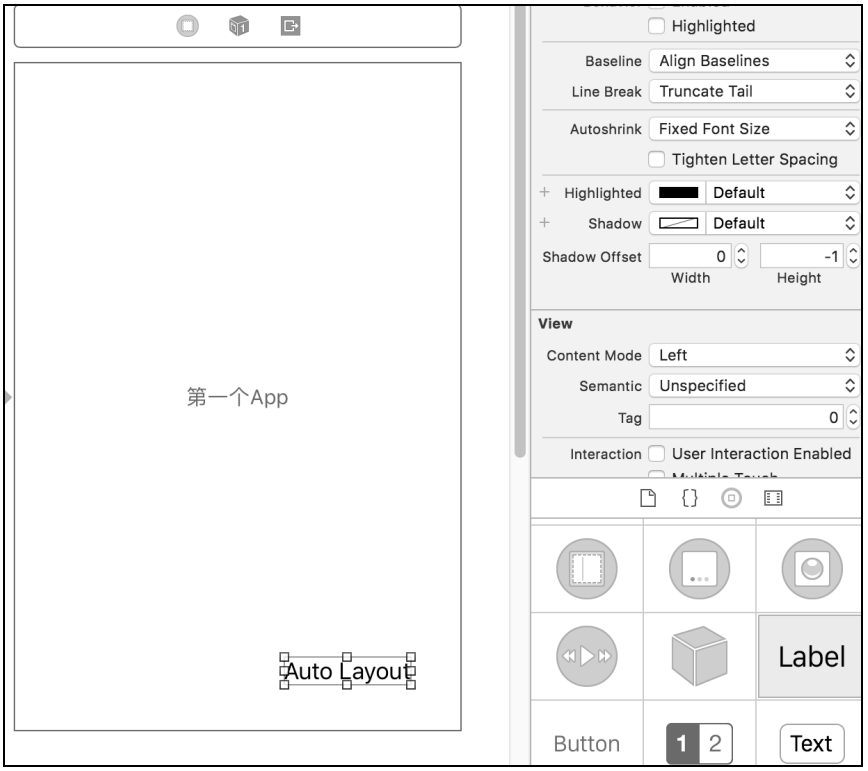

Highlighted
Baseline
Align Baselines
Line Break
Truncate Tail
Autoshrink
Fixed Font Size
Tighten Letter Spacing
Highlighted
Default
Shadow
Default
Shadow Offset
0
-1
Width
Height
View
Content Mode
Left
Semantic
Unspecified
Tag
0
Interaction
User Interaction Enabled
第一个App
Auto Layout
Label
Button
1
2
Text

图 22-1

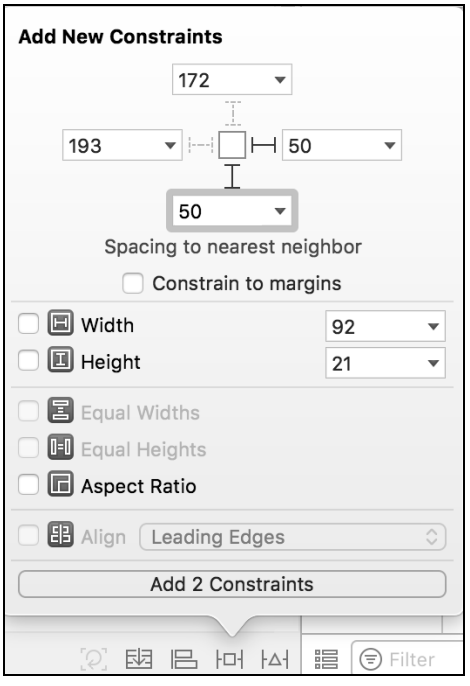

Add New Constraints
172
193
50
50
Spacing to nearest neighbor
Constrain to margins
Width
92
Height
21
Equal Widths
Equal Heights
Aspect Ratio
Align
Leading Edges
Add 2 Constraints
Filter

图 22-2

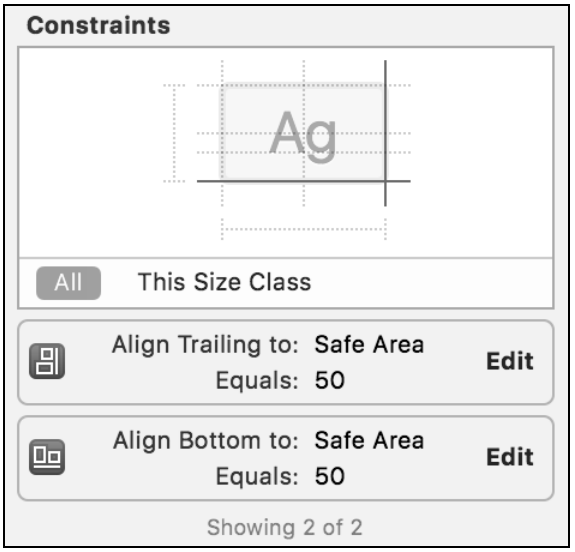

Constraints
Ag
All
This Size Class
Align Trailing to: Safe Area
Equals: 50
Edit
Align Bottom to: Safe Area
Equals: 50
Edit
Showing 2 of 2

图 22-3

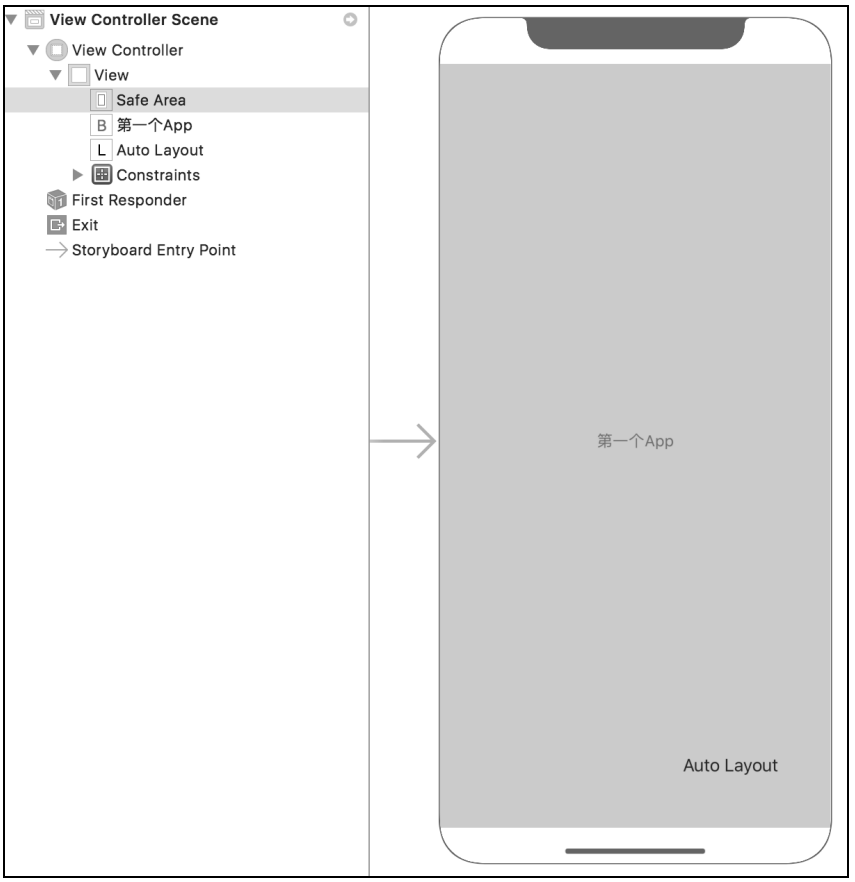

View Controller Scene
View Controller
View
Safe Area
第一个App
Auto Layout
Constraints
First Responder
Exit
Storyboard Entry Point
第一个App
Auto Layout

图 22-4

本节的作业是实现一个如图 22-5 所示的画面。



图 22-5

22.2 Stack View

对于用户来说，界面就是产品。本节要完成如图 22-6 所示的效果，有较多图文元素的场景所需的图片 t-shirts.zip 可以到 QQ 群 532084830 中下载。

对场景上的每个 UI 组件逐一定位显然烦琐、低效，灵活性差。各种主流移动平台都有不断优化的布局系统，以辅助开发者快速应付如今越来越复杂的各种排版样式。对于多个 UI 组件的整体定位，各平台的解决方案都相似，如表 22-2 所示。

iOS 的 Stack View 可作为一个容器，放入多个 UI 组件，容器内部有多种分配空间的方式可供选择，非常灵活。



图 22-6

表 22-2

平　台	容器式布局组件
HTML 5	Flexbox
Android	Constraint Layout
iOS	Stack View

Stack View 中的元素既可以垂直排列，也可以水平排列，所以有垂直和水平两种轴向（Axis）之分，如图 22-7 所示。

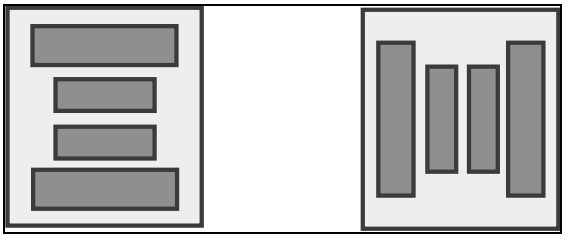


图 22-7

新建一个名为 StackViewDemo 的工程。工程文件中的 Assets.xcassets 是放置 App 运行过程中可能用到的各种多媒体资源的仓库。

左侧窗口是资源集列表，单击下方的+按钮，在弹出菜单中可以新增各种资源（图片集、颜色集、数据集、纹理集、立方体纹理、App 图标和启动图等），右侧窗口可预览资源。

我们知道，苹果产品一向在视觉上下足功夫，尤其是显示屏的细腻无颗粒感的效果首屈一指。iPhone 从第一代到 3s 都可以看出明显的颗粒感，因为屏幕上 1 个点，只显示 1 个像素。从 iPhone 4 开始一直到 iPhone 8，还是同样大小的 1 个点能显示 4 个像素，人眼已经分辨不出颗粒感了，乔布斯称之为 Retina 屏。普通屏与 Retina 屏显示效果比较，如图 22-8 所示。

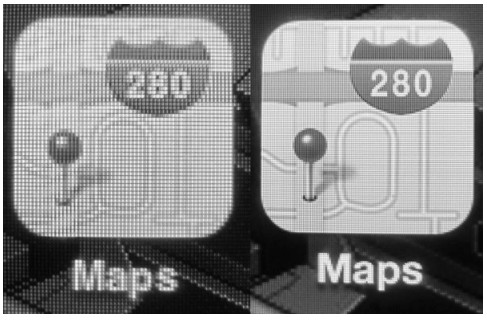


图 22-8

而 iPhone Plus 系列和 iPhone X 的 1 个点显示 9 个像素，显示效果更加细腻，也就是 Retina HD 屏。iPhone 各型号的屏的像素个数如表 22-3 所示。

表 22-3

iPhone	屏幕 1 个点含像素（个）
1/3/3s	1
4～8	4（Retina）
Plus，X	9（Retina HD）

这就对 App 的图片分辨率提出更高的要求，在 Retina 上显示的图片像素要多 4 倍，也就是长宽要翻倍。而在 Retina HD 屏上要多 9 倍的像素，长宽就要翻 3 倍。

这也就要求美工在制作 iOS App 上每一张图片时，同时要制备 3 种分辨率。

不过现在 App UI 设计已经非常成熟，这只是小菜一碟，可以一键导出。

把我们这种所需要的图片 t-shirts.zip 解压，每个图片有 3 种尺寸。文件名后缀的@1x、@2x、@3x 代表分辨率的倍数，如果@1x 的图片大小是 100×100 像素，那么@2x 是 200×200 像素，@3x 倍就是 300×300 像素，如表 22-4 所示。

表 22-4

文件名	分辨率（像素）
myImage@1x	100 × 100
myImage@2x	200 × 200
myImage@3x	300 × 300

加入图片到 Asset 最快捷的方法是把它们拖入资源集列表中，如图 22-9 所示。

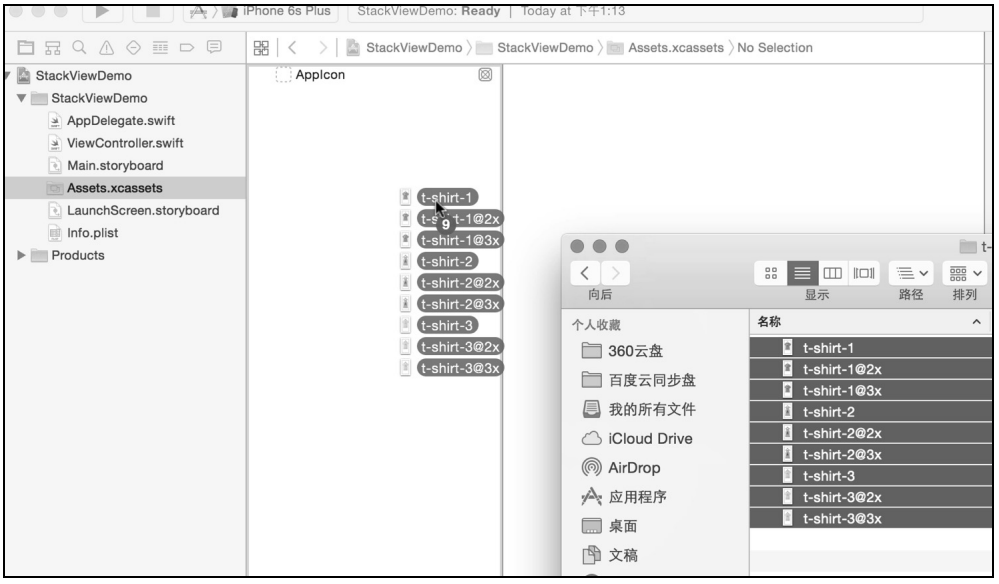


图 22-9

从组件库中拖曳一个垂直 Stack View 到场景上半部分，作为顶部文字的容器，容纳垂直排列的主标题和副标题，如图 22-10 所示。

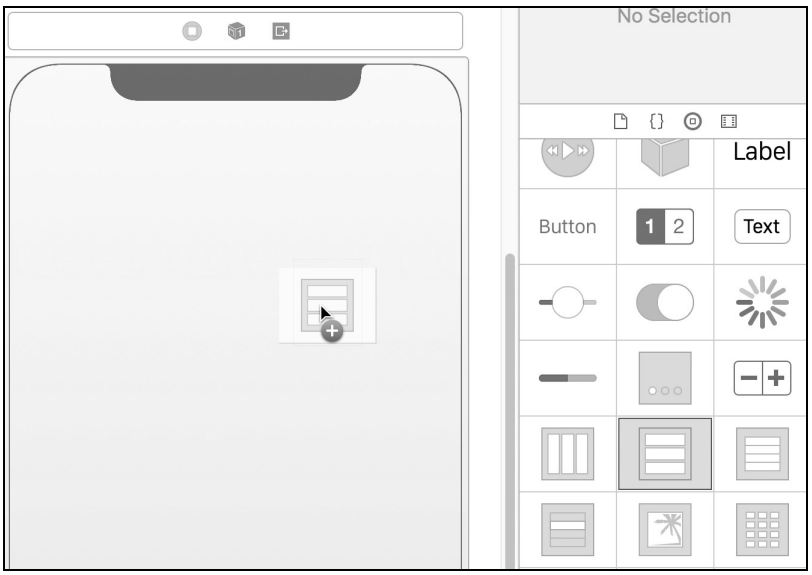


图 22-10

在这个垂直 Stack View 中拖入两个 Label 作为主标题和副标题，文字如图 22-11 所示，字号分别是 30 和 20。设定这个容器整体的约束：水平居中，顶距 40。



图 22-11

Stack View 的特殊之处是什么呢？看看它有哪些神奇的属性。

首先是轴（Axis）：垂直或水平在这可以随时切换，再也不用担心选错 Stack View 了。Stack View 的属性如图 22-12 所示。

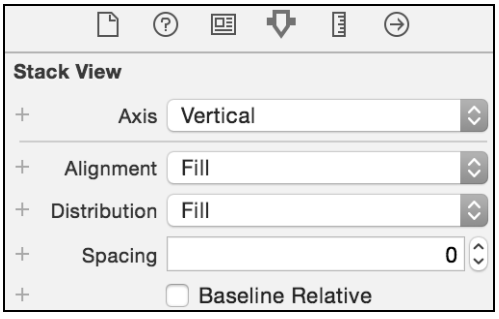


图 22-12

Alignment 属性用于设置内部项目在轴向上的对齐方式，垂直排列时有左、中、右对齐，水平排列则有顶、中、底对齐。这里设置为居中对齐（Center）。

Distribution 属性用于分配各项目在容器内占据的空间，除默认是按 UI 组件原大小分配外，还可以均等分配空间。

Spacing 则是各项目彼此的间距大小，默认值是 0。

接着拖曳 3 个 ImageView 到场景中部水平排列，单击约束栏第二个图标“Embed In Stack”，把它们组合到一个 Stack View 中，如图 22-13 所示。

因为组合之前是水平排列的，组合之后的 Stack View 默认就是水平轴向，这种使用 Stack View 的方法更直观。

为了使图片平均分配空间，将 Distribution 设为 Fill Equally。

设置 3 个 Image View 的 Image 分别为 t-shirt-1、t-shirt-2、t-shirt-3。整体约束为水平和垂直居中。图片设置如图 22-14 所示。

现在中部图片 Stack View 的问题是超出了屏幕宽度，如果能与屏幕等宽的话就好了。

这里介绍另一种“相对”的约束，一个 View 的大小可以参照另一个 View。

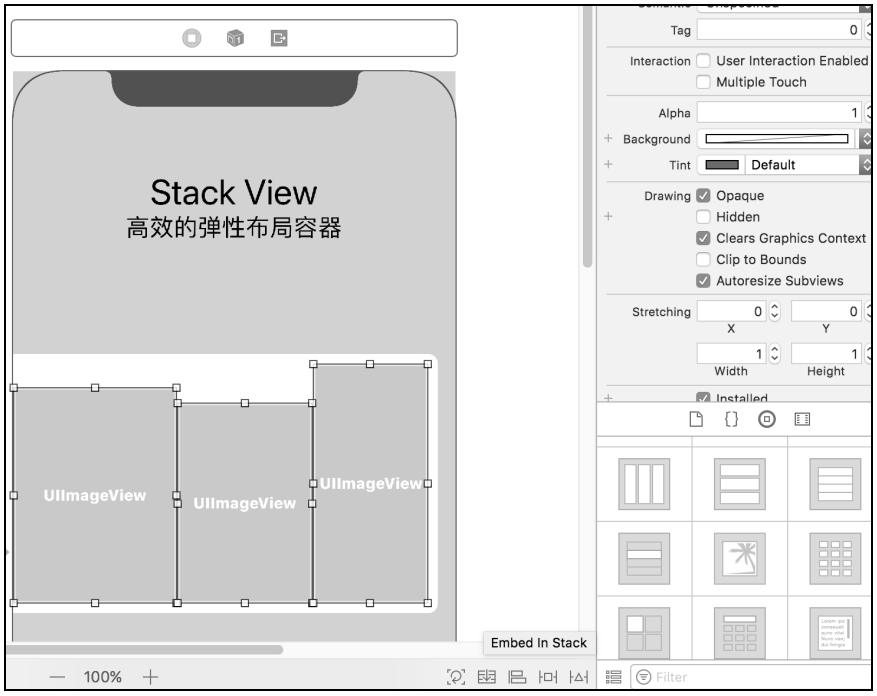


图 22-13

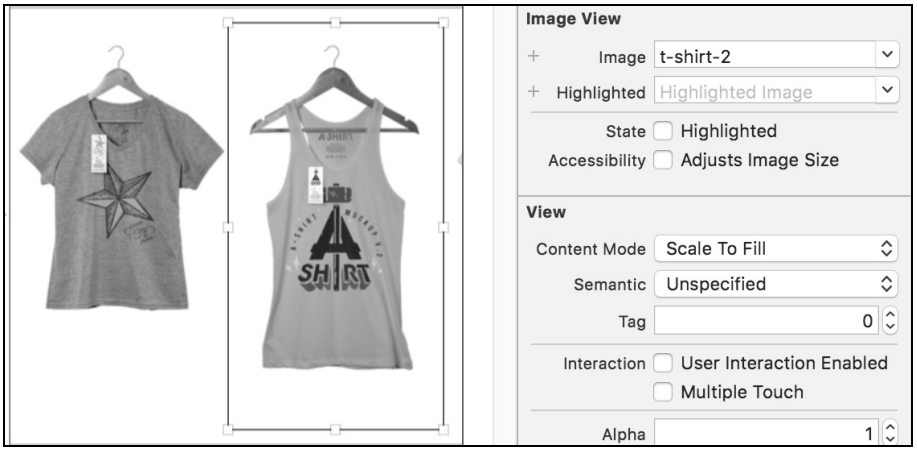


图 22-14

步骤如下。

（1）打开场景结构栏，按住【Ctrl】键拖曳图片 Stack View 到 Safe Area，如图 22-15 所示。

（2）放开鼠标则出现所有可行的约束菜单项供选择，选择 Equal Widths，即等宽，如图 22-16 所示。

调整后感觉好多了，但还存在一个问题，就是图片由于空间有限，被挤压发生变形了，如图 22-17 所示。

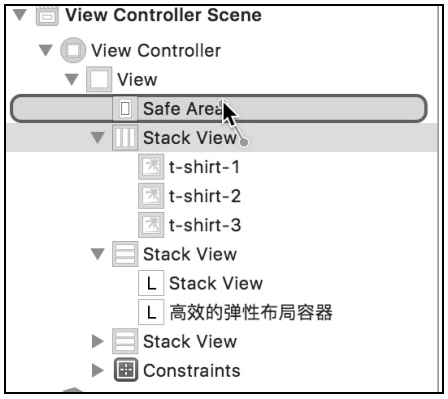


图 22-15



图 22-16



图 22-17

我们设置 Windows 桌面墙纸的填充方式时，有平铺、居中和拉伸三种方式。

这里的情况就是图片被拉伸了，查看 Image View 的 Content Mode 属性，默认是 Scale To Fill（拉伸充满容器），现改为 Aspect Fit（居中保持图片原比例）即可。

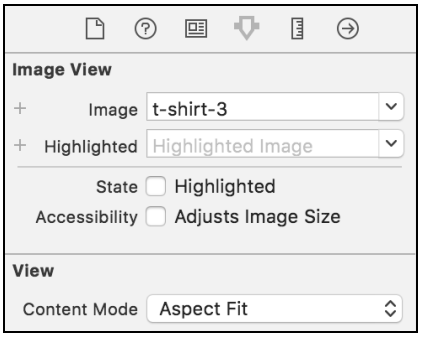


图 22-18

下面设置底部的按钮 Stack View。加入两个 Button，垂直轴向，居中对齐。按钮标题是“注册账号”和“第三方登录”。整体对外约束：水平居中，底距 30。按钮设置如图 22-19 所示。

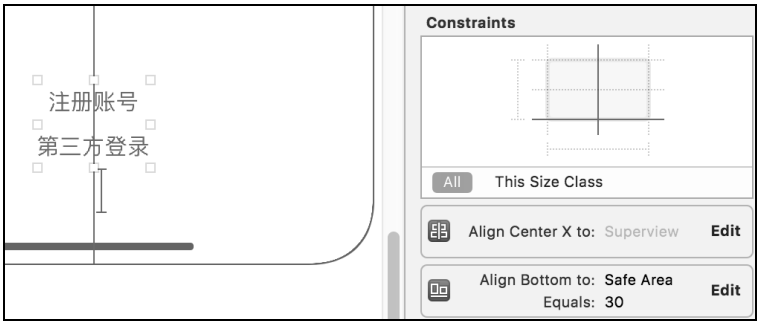


图 22-19

最终运行效果就如图 22-6 所示。

最后留给读者一个问题，如果把模拟器横屏后，会发现布局发生问题，这是因为此时屏幕长、宽已经发生了对调，该如何解决呢？



图 22-20

22.3 Size Class

22.2 节结尾时留下一个思考题，在正常竖屏下布好局的界面，到横屏下因为长宽发生对调导致布局出现问题。

本节的名字“Size Class”字面意思是“尺寸分类”，不仅解决横、竖屏下布局的问题，还对不同尺寸的 iOS 设备屏幕尺寸做了分类。

所以本节内容以 22.2 节完成的工程为模板，带读者一探 Size Class 的奥秘。

实现如图 22-21 所示的效果，在横屏下，顶部文字 Stack View 变成水平轴向，顶距变小，底部按钮 Stack View 隐藏。



图 22-21

我们可能在 storyboard 设计的设备预览栏里发现，每个设备后面都有一个很奇怪的符号，如（wR hC）等，如图 22-22 所示，这到底是何意？

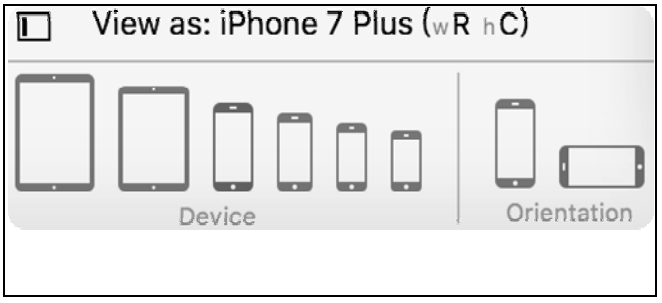


图 22-22

其实，这就是 Size Class 中最重要的术语，对屏幕尺寸的归类，其含义如表 22-5 所示。

表 22-5

高	宽	常规	紧凑
height	width	Regular	Compact

各种尺寸的 iPad 和 iPhone 的 Size Class 如表 22-6 所示。

表 22-6

	iPad	iPhone		
		竖屏	横屏	
h	R	R	C	
w	R	C	C	R（Plus）

这看起来好像很复杂，其实可以简单地归纳：iPad 的定位是小型计算机，无论何种 iPad 机型，无论横屏或竖屏，屏幕长和宽始终是足够 App 来布局的。而 iPhone 在正常竖屏使用时，因为要满足单手可以握住的特性，宽度始终是紧凑的（wC），再长的内容，哪怕是在很小的屏幕上也可以向下拖动，所以高度是足够的（hR）；在横屏状态下，长和宽都显得紧凑（hC wC），但 iPhone Plus 系列因为竖屏时足够高，所以横屏时宽度是足够用的（wR）。

记不住设备的 Size Class 也没关系，选中预览设备，其后面的那个符号就出现了。

在实战 Size Class 前，先打开场景结构栏，这里的相同 Stack View 太多了，不好区分。选中后单击名称即可修改显示名称，如“标题组”、“图片组”，“按钮组”，如图 22-23 所示。其实读者可能已经发现了，非容器类的 Button、Label、Image View 一旦设置了主要属性，就自动以主要属性为显示名称。

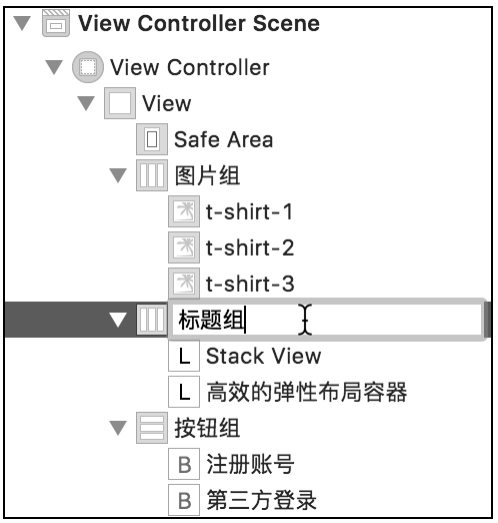


图 22-23

我们也可以在该视图的 ID 栏的 Label 属性中更改属性，如图 22-24 所示。



图 22-24

下面我们对标题组进行调整，先切换成 iPhone X 的横屏，Size Class 是（wC hC），可以代表除 iPhone Plus 系列以外所有的横屏情况。

选中标题组，单击其 Axis 属性坐标的加号，单击“Add Variation”按钮，添加一个在 wC hC 下的 Axis 属性“变体”，如图 22-25 所示。

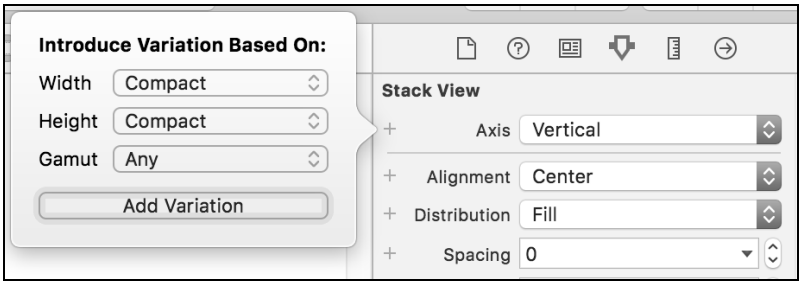


图 22-25

把这个新增的变体属性值改成 Horizontal，即在这种 Size Class 下，标题组的轴向是水平的。如法炮制，我们发现 Spacing 也可以调整，如变体值是 20，如图 22-26 所示。

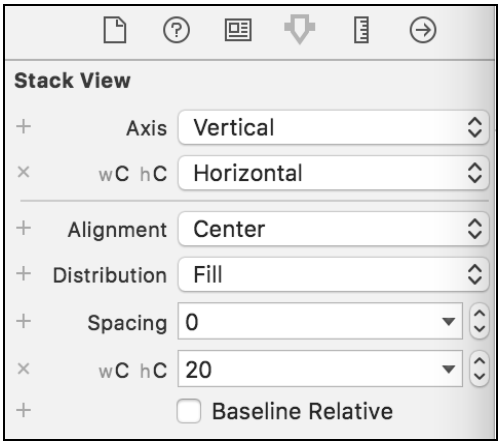


图 22-26

另外，各种约束也可以设置变体，如标题组的顶边距在 wC hC 下可以更小，找到顶边距约束，双击进入，给 Constant 增加 wC hC 变体，值为 10，如图 22-27 所示。

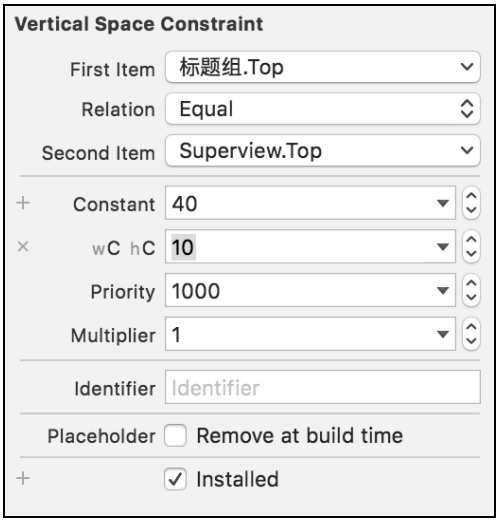


图 22-27

我们会发现设计预览上的变化，这时切换为竖屏，会发现竖屏的排版不受任何影响。

横屏有一个布局原则就是无关紧要的 UI 元素可以隐藏，如按钮组我们可能希望在 wC hC 下隐藏。选中按钮组，增加 Hidden 属性变体，勾选 Hidden 复选项，如图 22-28 所示。

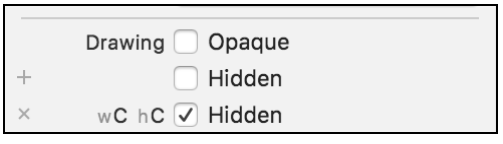


图 22-28

运行效果，如图 22-29 所示。



图 22-29

这时切换为竖屏，发现布局一切正常，非常神奇，至此我们没有输入一行代码。

横屏下还可以把图片组与 Safe Area 的等宽约束进行调整，但对于约束，目前只能对 Constant 增加变体，Multiplier 并不能增加变体。

有一种变通的方法，即先把等宽约束增加一个 Installed 在 wC hC 下的变体，取消勾选后面的 Installed 复选项，这样 1∶1 的等宽约束在横屏下就不生效了，如图 22-30 所示。

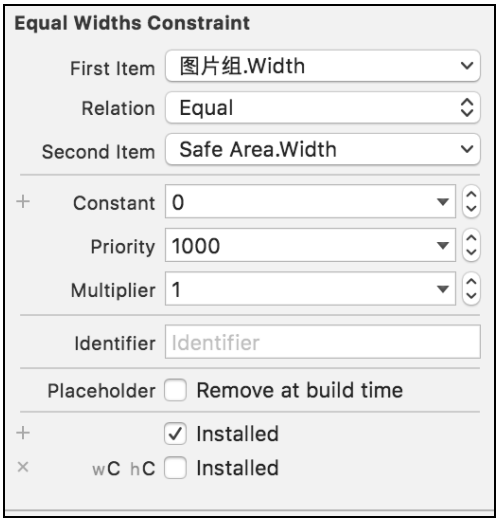


图 22-30

手工再增加一个等宽约束，系统调整为 0.8。增加 wC hC 变体，勾选后面的 Installed 复选项，并取消勾选默认的 installed 复选项，即 0.8 的等宽约束，只在 wC hC 下起效，如图 22-31 所示。

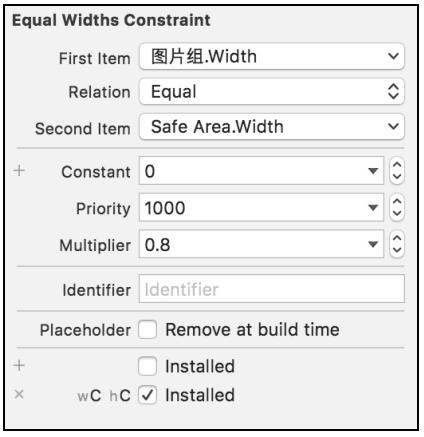


图 22-31

这样就变通地达成了同一种约束中的系数，在不同 Size Class 下可以有不同的值，将来的 XCode 版本，可能会直接给 Multiplier 增加变体。

至此，相信读者已经了解了 Size Class 的强大和便利之处。

在本节一开始，我们提到了 iPhone 横屏时有两种 Size Class，本节没有介绍 iPhone Plus 横屏（wR hC）的情况，请读者自己尝试调整布局。

22.4　Table View

本节我们来到 iOS 开发中可能是最常用，也是最重要的一个 UI 组件——Table View。

Table View 用于创建各种列表型页面，如微信聊天列表（见图 22-32）、支付宝的上映电影列表（见图 22-33）、App Store 首页的 Today 卡片式列表（见图 22-34）。



图 22-32



图 22-33



图 22-34

Table View 的特点是仅显示一列，如果一定要在一屏内显示不同类型的列表，可以用“块（Section）”来分隔。例如，在 iOS 系统设置中，把功能分区隔离，如图 22-35 所示。



图 22-35

本节的目标是实现一个固定的列表 App，带有图片介绍的绝地求生武器介绍，完成效果如图 22-36 所示。

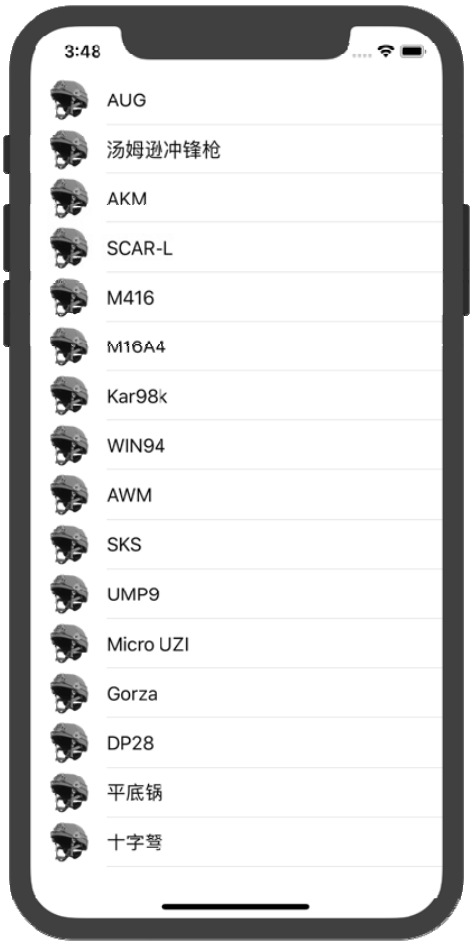


图 22-36

本节内容主要分三部分：使用 Table View 组件及配置原型单元格；介绍 iOS 开发最常用的开发模式——代理模式；把数据源通过代理连接到 Table View。

22.4.1　原型单元格

新建工程 Table View Demo，从组件库拖曳一个 Table View 到主场景上，如图 22-37 所示。设置约束为四个边距为 0，如图 22-38 所示。

Table View 既然是用于批量显示数据的，那么一定有显示模板。这样的模板称为“原型单元格”，即定义好这个单元格的布局，根据业务需要把数据填充到不同的原型单元格上。

选择 Table View，调整其属性 Prototype Cells 为 1，因为目前需求很简单，只需要一种模板，如图 22-39 所示。

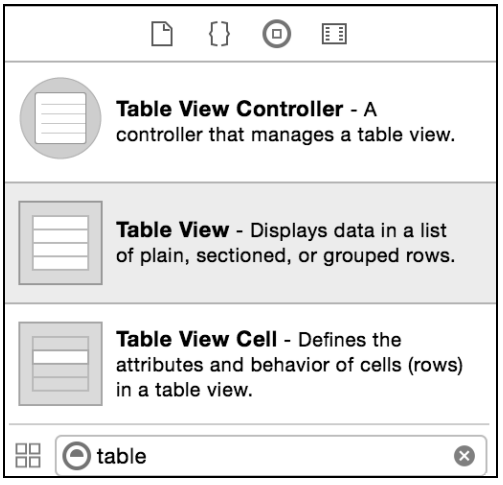


图 22-37

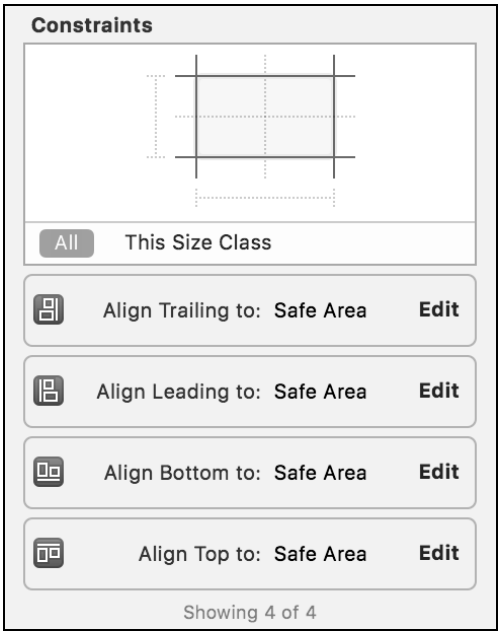


图 22-38

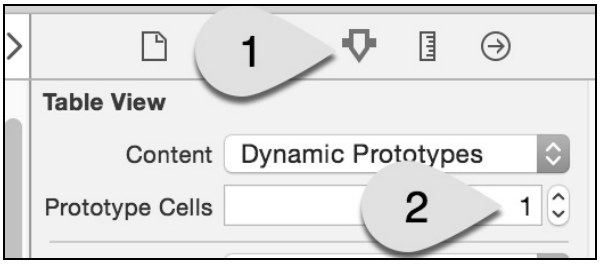


图 22-39

那么这个单元格的样式要如何定义呢？除自定义样式外，有时为了简单起见，也可以直接使用预置的样式。打开场景结构栏，选择 Table View 下的 Table View Cell，将其属性中的 Style 设置为 Basic，这时在属性栏会多一个 Image 属性，然后场景上会显示一个 Title。Basic 如设计预览所示，只显示一个 Label，并可以带一张图。Identifier 就是给模板起个名字：WeaponCell，以便重复使用时这个模板容易被识别。原型单元格属性设置如图 22-40 所示。

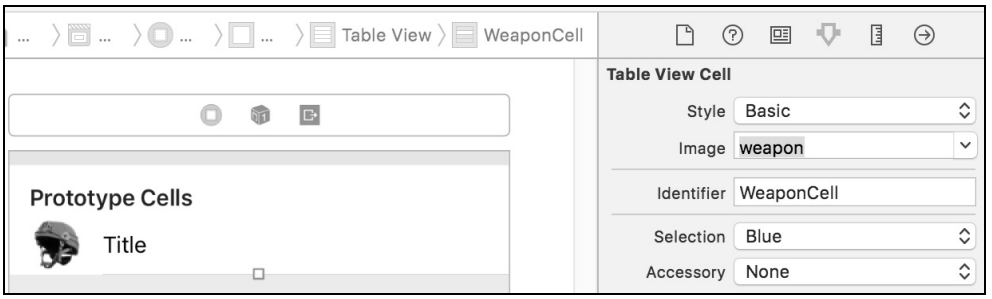


图 22-40

把本节附带的 weapon.zip 解压加入 Assets.xcassets 中，然后就可以在 Cell 的 Image 属性中选择这个图片了。

现在运行一下，什么内容都没有。

22.4.2　代理模式

我们知道，App 的数据一般都来自本地文件或 App 服务器，随着时间会动态发生变化，这些数据如何与 UI 组件结合起来，这就需要了解软件开发中的“设计模式”了。所谓“设计模式”，通俗来讲就是一些写代码的“套路”。

苹果在 iOS SDK 中大量使用了一种“设计模式”，即“代理模式”。尤其在控制 UI 相关的组件中，之前我们用的几乎所有 UI 组件都位于 UIKit 这个框架中，Label 和 Button 对应的类是 UILabel 和 UIButton，Table View 对应的类是 UITable View。

对于 Label 和 Button 这种相对简单的 UI 组件，Table View 的控制要复杂一些，Table View 通过代理模式，把要显示的数据，代理给“别人”去完成。代理者只要遵守相关协议就可以了。

现在的 Table View 位于主场景上，View Controller 就成了理所当然的代理者。

实现代理协议也很简单，打开 ViewController.swift，在父类 UIViewController 后加上遵守协议 UITableViewDataSource，用逗号分隔，如图 22-41 所示。

这时 XCode 会提示错误，指出 View Controller 不遵守协议，因为没有实现协议规定的东西。单击红色空心小点，再单击“Fix”按钮。XCode 会自动补齐要实现的代码块。

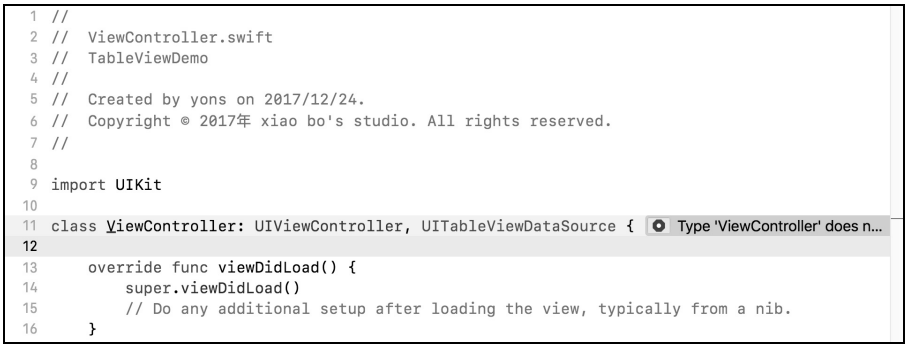


图 22-41

```
    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int)
->Int {

        }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
->UITableViewCell {

        }
```

第一个待补齐的方法是指 Table View 中每个块（Section）中显示多少行。还记得一开始时我们讲过，如果一个列表里有类型不同的数据，可以分成不同的列表块。Table View 默认只有一个块。

目前我们的数据还没有涉及网络数据，所以用一个 View Controller 的属性来保持数据，这就好像是离线数据：

```
    var weapons = ["AUG","汤姆逊冲锋枪","AKM","SCAR-L","M416","M16A4","Kar98k",
"WIN94","AWM","SKS","UMP9","Micro UZI","Gorza","DP28","平底锅","十字弩"]
```

有了数据，就可以给第一个方法补齐代码了：

```
    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int)
->Int {
    returnweapons.count
        }
```

第二个待补齐的方法是指每个块的每行的单元格是什么，在 22.4.1 节中我们设置了单元格为系统预置的 Basic 样式。在这个方法里，我们就可以把数据依次传给单元格上的 Label 了。我们在第一个方法指定了 Table View 有 16 行，那么第二个方法遇到每行时都会执行一次，也就是 16 次。我们如何知道究竟到 Table View 的哪个位置呢？这个位置信息保存在方法的参数 indexPath 里，indexPath 包含两个属性，section 和 row。indexPath 和数组一样，都是从 0 开始计数的。

那么 dequeResuableCell 是什么呢？如果我们这时看本书的配套视频（iOS Ⅱ开发第一季），可以看看视频中的演示动画。

dequeResuableCell 这一句是指从被回收的 Cell 队列中取回一个，重新作为原型来使用。为何回收 Cell 呢?

在本节开始时，我们提到 Table View 做过性能优化，因为手机内存有限，一次性画 100 个单元格不如重复使用 10 个单元格更快，用户滑动时的体验也更好。

到底重用哪种原型 Cell 呢？在 Table View 中定义原型 Cell 时留下的 identifier 来区分。

使用快捷键 Commmand + R 运行，依然没有数据，这是怎么回事呢？

22.4.3　连接数据源

虽然主场景控制器实现了 Table View 数据源协议指定的方法，但有一个问题，我们并没有让 Table View 指定它的数据源代理者是主场景。

要指定数据源代理者是主场景很简单，打开场景编辑器，按住【Ctrl】键拖动 Table View 到 View Controller，在出现的菜单上选择 dataSource 即可，如图 22-42 和图 22-43 所示。

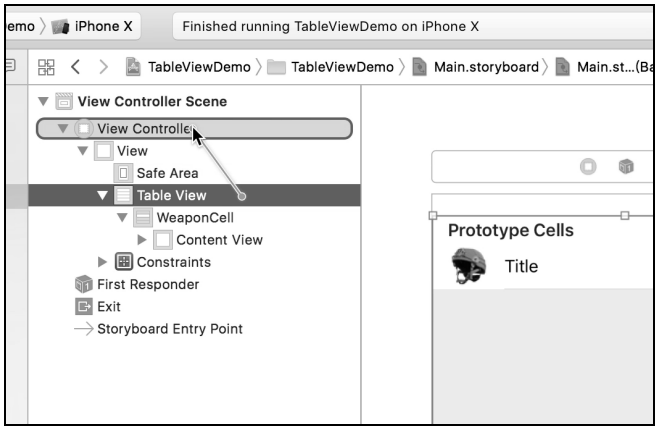


图 22-42

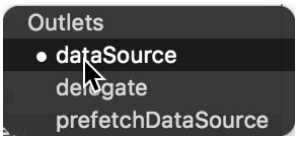


图 22-43

在 Table View 上右击，可以用来确定是否连接成功，如图 22-44 所示。

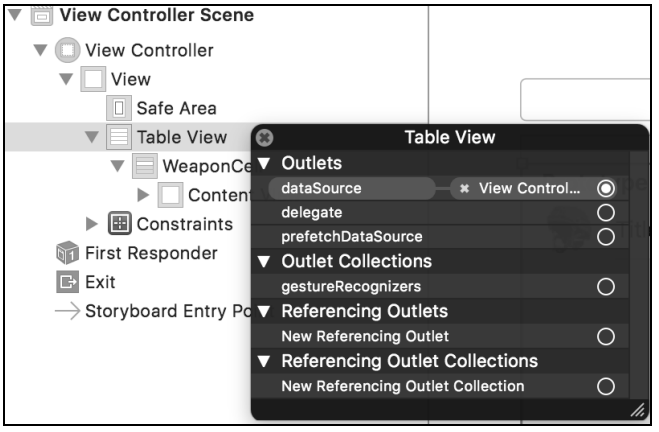


图 22-44

此时运行模拟器，可使用快捷键 Command+ R，看看最终的成果吧。

小结：Table View 是 iOS 开发中最常用的 UI 元素，理解本节内容非常重要。

作业：尝试使用每个单元格换一张图片。

22.5 自定义单元格

22.4 节我们对 iOS 中最常用的 Table View 组件做了初探，相信读者通过这个常规的组件对 iOS 开发模式有了初步的印象，这是一个很好的开始。不过由于使用了系统预置的 Basic 样式，说实在的是真丑。

从本节开始，我们要开始“叱咤风云”了。

先来看看本节要实现什么样的排版效果，如图 22-45 所示。



图 22-45

独眼帅气大叔、霸气外国小姐姐，马上就让人产生想点击的冲动。喜欢的图片还可以单击卡片右上角的爱心图标进行收藏。

本节分为四部分：UITableViewController 的使用；自定义单元格布局；Table View 数据填充；实现圆角和阴影效果。

1. UITableViewController 的使用

在入门级我们讲了 App 由各个页面场景构成，场景由视图组成，视图由控制器控制与

用户的交互。Table View 作为经常使用的一个组件，系统提供了一个控制器场景，其优点是自带 Table View，代理自动配置，无须布局。只要在组件库中选中并直接拖到 storyboard 上即可，非常方便。

新建一个工程，命名为 Weapons，拖曳一个 UITableViewController 到 Main.storyboard 上，把原入口场景的大箭头拖动到新场景上，如图 22-46 所示。

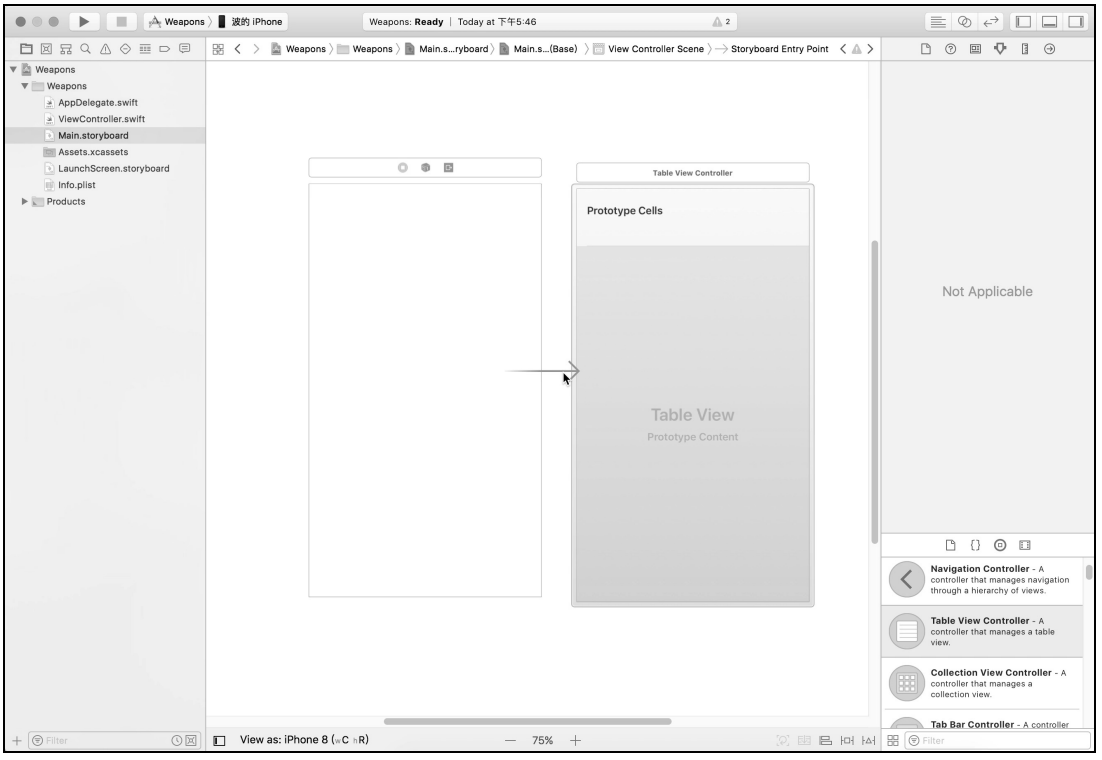


图 22-46

打开场景结构，选择原来的 View Controller Scene，按【Delete】键将它删除，并在工程文件中把对应的控制器 ViewController.swift 也一并移除，单击“Move to Trash”按钮以彻底移除出工程，如图 22-47 所示。

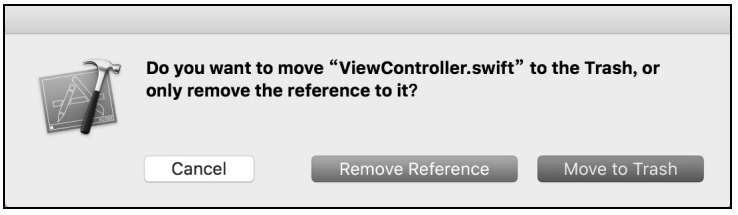


图 22-47

新建的场景需要想要的控制器，在任意工程文件上右击，在弹出的菜单中选择“NewFile…”来新建一个控制器，如图 22-48 所示。

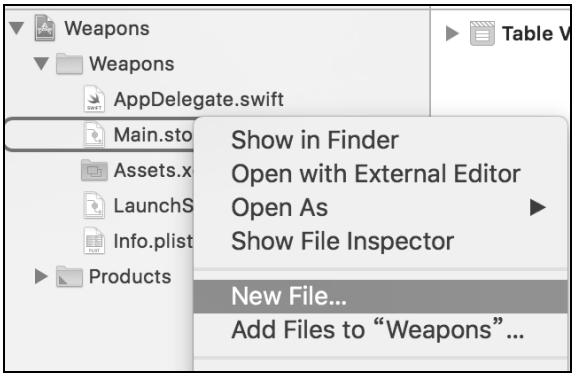


图 22-48

在文件模板中选择 iOS 下的 Cocoa Touch Class，单击“Next”按钮，如图 22-49 所示。

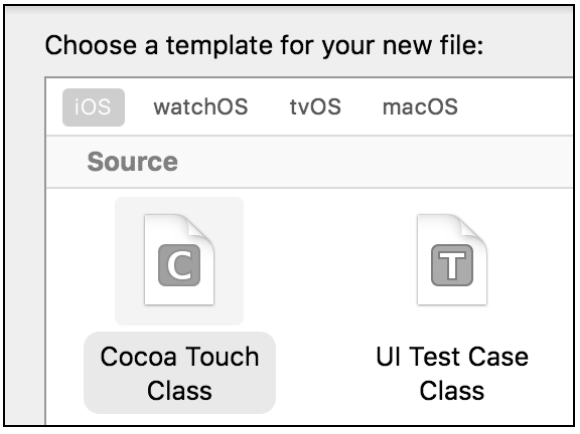


图 22-49

在控制器的 Subclass of（父类）一栏输入 UITableViewController，然后在 Class 栏输入 WeaponsTableViewController，如图 22-50 所示。

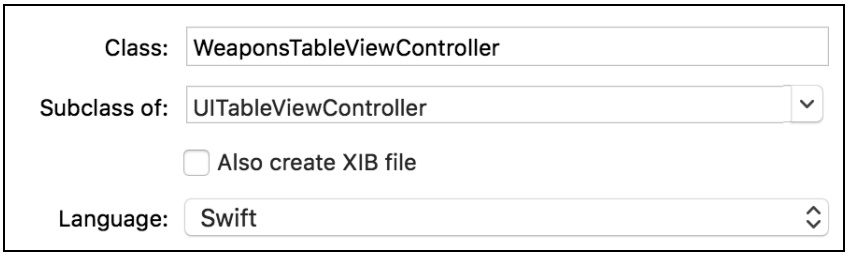


图 22-50

单击“Next”按钮后出现文件保存位置窗口，默认就在工程目录下，单击“Create”按钮。

此时场景和控制器都有了，需要把两者连接起来，在 Main.storyboard 中打开场景结构，

选中场景，在 ID 栏的 Class 属性中，单击下拉箭头，选中刚刚新建的控制器类名，即 WeaponsTableViewController。至此，新场景有了对应的控制器，接下来就可以开始实现布局和代码了。

我们讲过 Table View 需要给它提供数据源，下面我们给它提供 4 个数据源。

第 1 个，武器名称，在 WeaponsTableViewController.swift 加入：

```
    var weapons = [
    "AUG","AWM","十字弩","DP28","Groza","Kar98k","M16a4","Micro UZI","平底锅
","SKS","UMP9"
        ]
```

第 2 个，武器类型：

```
    var weaponTypes = ["自动步枪", "狙击枪", "冷兵器", "机枪", "自动步枪", "狙击枪
", "自动步枪", "冲锋枪", "近身武器", "半自动步枪",  "冲锋枪"]
```

第 3 个，武器原厂地：

```
    var origins = ["奥地利", "英国", "中国", "苏联", "俄罗斯", "德国", "美国", "以
色列", "美国", "苏联", "美国"]
```

第 4 个，武器对应的图片名称：

```
    var weaponImages = [
    "aug", "awm", "crossbow",
    "dp28", "groza", "kar98k",
    "m16a4", "microuzi", "pan",
    "sks", "ump9"
        ]
```

最后，武器图片见 QQ 群：532084830，weapons.zip，将它下载并解压后拖到 Assets 中。我们要强调一点，其实在本书中，所有涉及的资源图片，并非必需的，所有图片，读者完全可以自己在网上搜索并替代！

2. 自定义单元格布局

自定义单元格的效果如图 22-51 所示。

现在我们来对自定义单元格的组成做一个分解，首先卡片式单元格的高度足够大，设定默认的原型单元格高度为 500，如图 22-52 和图 22-53 所示。

场景结构依次分解，最外层是容器，用于包含模糊特效的标题栏、背景图、右下角的小标题。从组件库中拖曳一个 View 到 Cell 上，显示名称改为 CardContainer，约束上、下、左、右都为 10 如图 22-54 所示。

再拖曳一个 ImageView 到容器上做背景图，约束上、下、左、右都为 0，效果如图 22-55 所示。

标题栏其实也是个容器，但要实现模糊特效怎么做？iOS 有个很好用的组件，可以实现容器加模糊特效二合一，如图 22-56 所示，在组件库中找到 fx 图标，拖到 ImageView 上，填充模式为 AspectFill。约束上、左、右为 0，高度为 80，如图 22-57 所示。



图 22-51

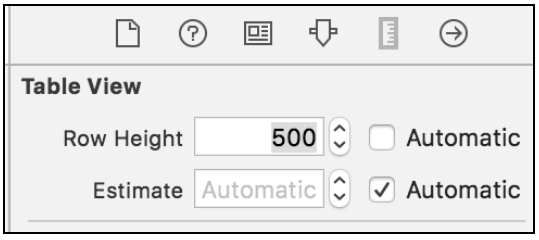


图 22-52

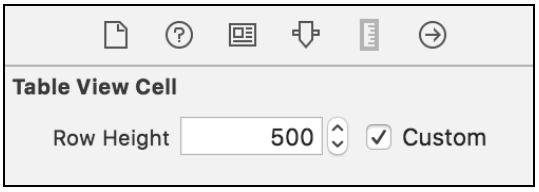


图 22-53

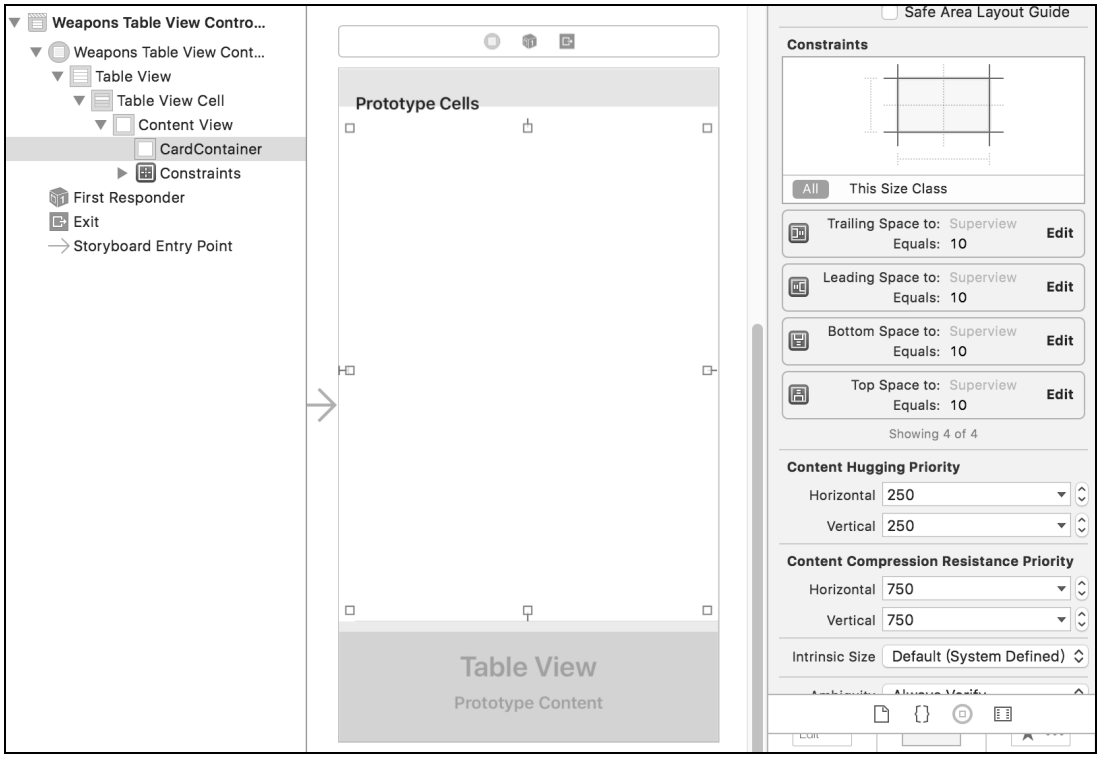

Weapons Table View Contro...
Weapons Table View Cont...
Table View
Table View Cell
Content View
CardContainer
Constraints
First Responder
Exit
Storyboard Entry Point
Prototype Cells
Table View
Prototype Content
Safe Area Layout Guide
Constraints
All
This Size Class
Trailing Space to: Superview
Equals: 10
Edit
Leading Space to: Superview
Equals: 10
Edit
Bottom Space to: Superview
Equals: 10
Edit
Top Space to: Superview
Equals: 10
Edit
Showing 4 of 4
Content Hugging Priority
Horizontal 250
Vertical 250
Content Compression Resistance Priority
Horizontal 750
Vertical 750
Intrinsic Size Default (System Defined)

图 22-54

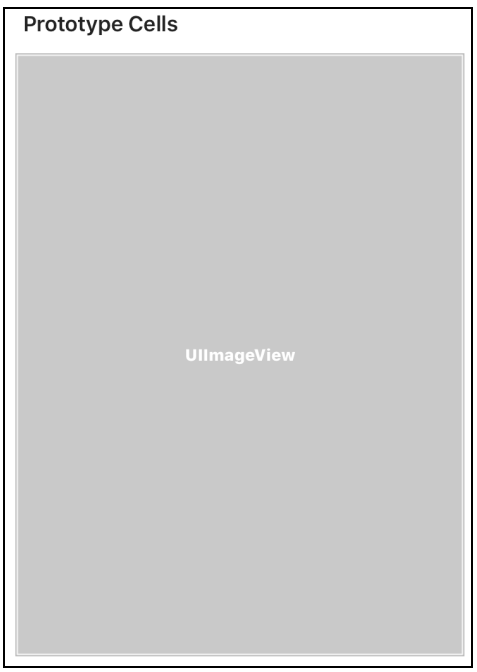

Prototype Cells
UIImageView

图 22-55

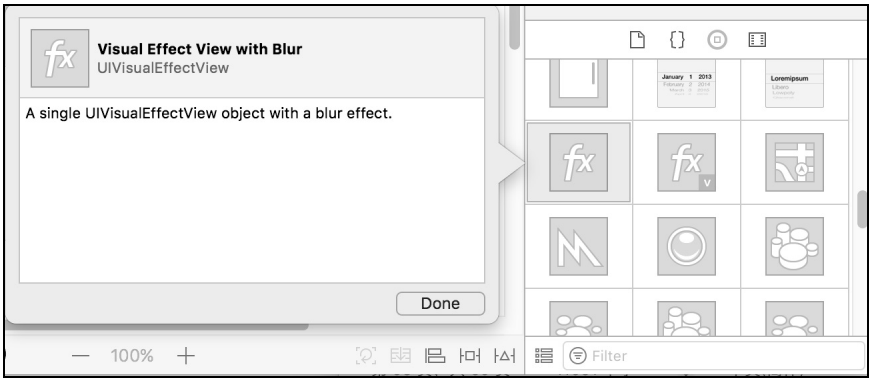


图 22-56

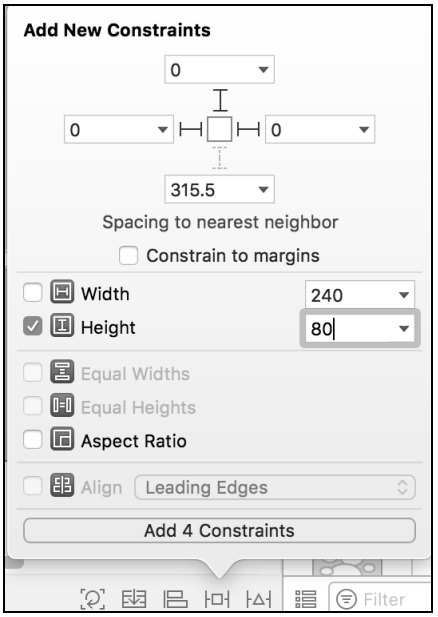


图 22-57

容器内左侧增加两个 Label 做大小标题文字，第二个 Label 字号为 25，组合成 Stack View，约束为左边距 20，垂直居中。右侧为一个按钮，图片为 unfav（空爱心），约束为垂直居中，右边距为 20，长、宽均为 50。布局效果如图 22-58 所示。

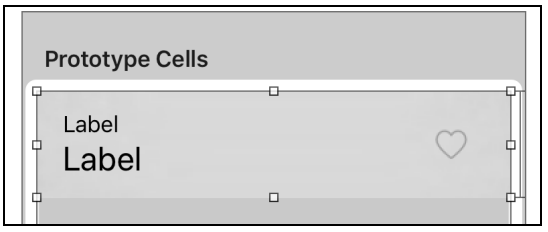


图 22-58

右下角 Label 的两个约束也都是 20，如图 22-59 所示。

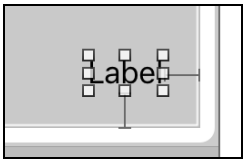


图 22-59

不要忘记将单元格的 Identifier 设为 CardCell。

在入门级解析场景结构时我们讲过，每个子视图都可以单独有控制器，因为这里已经自定义了单元格，所以我们需要一个控制器。选择菜单 NewFile→CocoaTouchClass，父类是 UITableViewCell，Class 命名要与单元格的 Identifier 一样都是 CardCell，再连接起来，如图 22-60 所示。

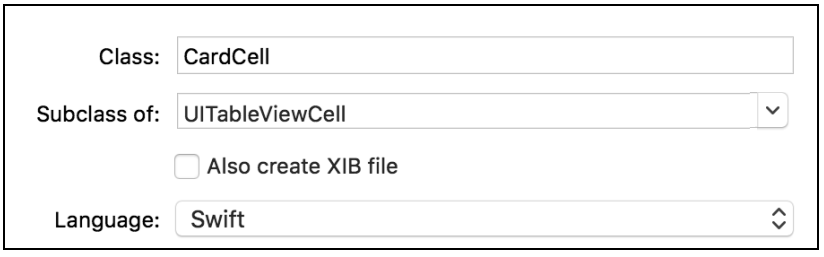


图 22-60

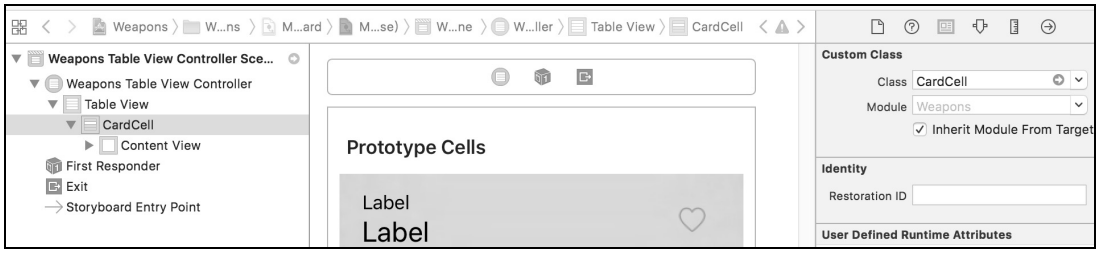


图 22-61

单击右上角的双环按钮，同时打开界面设计器和代码编辑器，在场景结构中选中 CardCell，代码编辑器应会自动切换到它的控制器，不过这种概率是 20%，这也是 XCode 的一个老漏洞了。如果切换不出来的话，就在左上角选择 Manual，再选择 CardCell.swift。把单元格上的几个 UI 组件都与控制器连接：

```
@IBOutletweakvar weaponLabel: UILabel!
@IBOutletweakvar favBtn: UIButton!
@IBOutletweakvar originLabel: UILabel!
@IBOutletweakvar typeLabel: UILabel!
@IBOutletweakvar backImageView: UIImageView!
```

3. Table View 数据填充

现在自定义样式已经确定，下面把数据填充到 Table View 上。还记得数据源代理方法实现吗？只要继承 UITableViewController，代码会自动实现数据源的方法。因为 Table View 默认是 1 个 Section，所以删除默认 numberOfSections 方法。

实现 Table View 中有多少行的方法：

```
    override func tableView(_tableView: UITableView, numberOfRowsInSection
section: Int) ->Int{
            return weapons.count
    }
```

把 cellForRowAt 那段已经注释掉的代码解除注释（XCode 为 UITable View Controller 自动生成的），代码如下：

```
    overridefunc tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) ->UITableViewCell {
    let id = String(describing: CardCell.self)

    let cell = tableView.dequeueReusableCell(withIdentifier: id, for: indexPath)
as! CardCell

            cell.typeLabel.text = weaponTypes[indexPath.row]
            cell.weaponLabel.text = weapons[indexPath.row]
            cell.originLabel.text = origins[indexPath.row]
            cell.backImageView.image = UIImage(named: weaponImages[indexPath.
row])

    return cell
        }
```

以上代码中用 String 的一个构造器方法来获取单元格的 Identifier，为什么不直接写“CardCell”呢？因为直接写很有可能因为拼写错误而造成运行错误。

下一句代码就是重用单元格，只不过后面加了个 as! CardCell，意思是强制转换为 CardCell，因为这里都是我们自定义的单元格 CardCell。

后面的 4 句代码就是把数组中相应的项目填充到相应的 UI 元素上。

运行效果如图 22-62 所示，圆角和阴影待实现。

先把两个细节优化了，如每个单元格间的分隔线，把 Table View 的 Separator 属性设为 None 即可。右下角 Label 字体改为白色，加上阴影，阴影偏移是（1，1），这样在深色背景上更易分辨。

4. 实现圆角和阴影效果

谈到图片效果，貌似都是美工的事情，不过就算有美工，做这件事也要靠工具。每个视图除有子视图外，还可以像 Photoshop 中由更细致的图层来构成，常用的图层属性有边

角半径、阴影色及偏移、阴影透明度、边框粗细和颜色等。不过可惜的是，现在还不能像 Photoshop 那样可以直接通过属性栏设置，而是要写代码。



图 22-62

虽然目前不能直接设置这些属性，但也不用每次都写这些代码，利用 Swift 语法的属性扩展及 XCode 的 IBInspectable 特性，可以只写一次代码，在属性栏中直接设置。

新建一个 UIViewHelper.swift，解决圆角设置，代码如下。

```
import UIKit

extensionUIView {
  @IBInspectable
    var cornerRadius: CGFloat {
    get {
      returnlayer.cornerRadius
```

```
    }

    set {
      layer.cornerRadius = newValue
    }
  }
}
```

这时选中任何一个 View，在属性栏的上部都可以设置圆角半径，如图 22-63 所示。



图 22-63

设置阴影圆角和透明度，代码如下。

```
@IBInspectable
  var shadowRadius: CGFloat {
    get {
      returnlayer.shadowRadius
    }

    set {
      layer.shadowRadius = newValue
    }
  }
@IBInspectable
 var shadowOpacity: Float {
    get {
      returnlayer.shadowOpacity
    }

    set {
      layer.shadowOpacity = newValue
    }
  }
```

设置阴影色，代码如下。

```
 @IBInspectable
 var shadowColor: UIColor? {
    get {
```

```
            returnlayer.shadowColor != nil ? UIColor(cgColor: layer.shadowColor!) :
nil
        }

        set {
          layer.shadowColor = newValue?.cgColor
        }
      }
```

设置阴影偏移，代码如下。

```
     @IBInspectable
     var shadowOffset: CGSize {
        get {
          returnlayer.shadowOffset
        }

        set {
          layer.shadowOffset = newValue
        }
      }
```

CardContainer 的设置如图 22-64 所示。

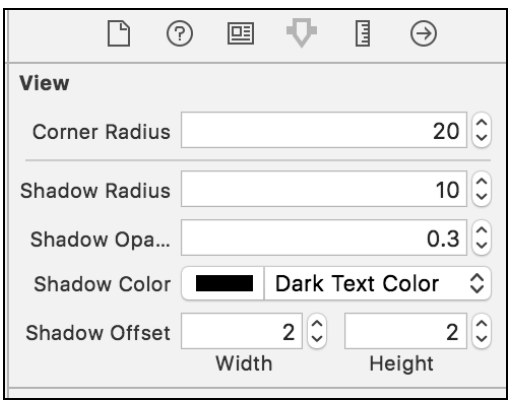


图 22-64

背景图和标题栏除设置圆角半径与 CardContainer 相同外，还要勾选裁边属性（Clip To Bounds），因为裁边后才能显示圆角，如图 22-65 所示。



图 22-65

作业：尝试实现爱心按钮单击的变化。

22.6 单元格交互

本节详细介绍单元格内的组件，以及如何与用户交互。

1. MVC 模式

首先总结一下我们到现在为止的“设计模式”，之前已经介绍了代理模式。视图、视图控制器和数据三者之间的关系，可以用一个简单的模式来称呼——MVC，如图 22-66 所示。

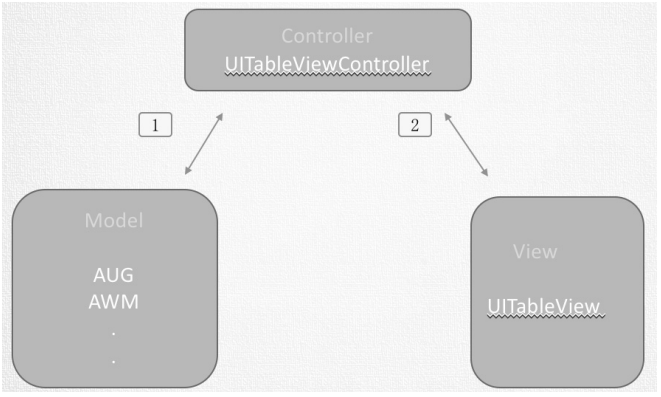


图 22-66

物以类聚，人以群分，数据也需要进行分类，分类好的数据称为数据模型（DataModel），简称 Model。用户看到的都是视图（View），视图的控制器（Controller）负责数据和视图之间的交互，从而实现数据与画面的分离。MVC 并不是什么新概念，早在乔布斯和比尔·盖茨的操作系统 Mac 和 Windows 出现以前的 Unix 时代，就已经有人发明了 MVC 模式。MVC 模式让一个程序既可以有命令行版本，也可以开发一个有界面的版本，这让普通用户也可以顺利使用。

下面以爱心按钮的交互为例，实战 MVC。

Model：保存武器的收藏状态。收藏只有两种状态，单个单元格（CardCell）需要保存收藏与否的状态，可以用一个布尔变量来表示：

```
var favorite = false
```

主场景控制器（AreaTableViewController）需要保持所有武器的收藏状态，用一个布尔型数组来保存：

```
var favorites = Array(repeating: false, count: 11)
```

View：收藏按钮。单击后需要切换按钮的图片。

Controller：主场景控制器。

2. 爱心按钮交互

接着就是创建用户交互了。不知道读者是否还记得创建按钮的用户交互步骤。没错，

同时打开界面构建器和代码编辑器，按住【Ctrl】键拖曳爱心按钮到代码中，命名为favBtnTap，如图 22-67 所示。

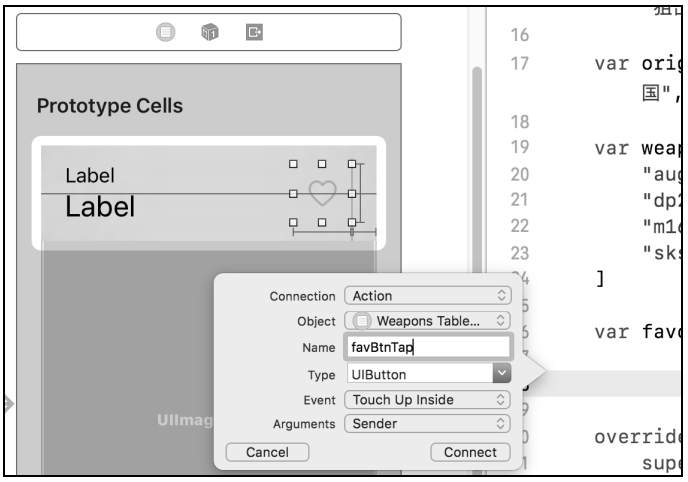


图 22-67

接下来完善代码。单击该按钮之后，首先想办法知道当前是在哪一个单元格，才能知道是哪个武器，从而更新收藏状态。代码如下。

```
@IBActionfunc favTap(_ sender: UIButton) {
let btnPos = sender.convert(CGPoint.zero, to: self.tableView)
let indexPath = tableView.indexPathForRow(at:btnPos)!

self.visited[indexPath.row] = !self.visited[indexPath.row]

let cell = tableView.cellForRow(at: indexPath) as! CustomTableViewCell
      cell.visited = self.visited[indexPath.row]
   }
```

第一步，把按钮自身的位置坐标转换成 table View 中的坐标，然后从坐标获取 index path，有了位置就可以更新武器收藏数组中该武器的状态了。

最后还要更新当前单元格的数据状态。

现在运行的话，只是数据更新了，但爱心按钮图片还没有更新，它可以在单元格的控制器中更新。这里再给读者介绍一个新的设计模式，称为观察者模式（Observer Mode）。观察者模式是指给一个数据加上观察者，一旦数据发生变化，观察者会立即发出通知，就好像被间谍盯上一样。

爱心按钮的变化与单元格的收藏与否状态密切相关，恰好 Swift 语法提供了一个观察者模式的应用——属性监视器。

给一个属性加上监视器很简单，只需在属性定义后加上一对花括号。属性监视器提供两个状态：一个是属性即将发生变化（willSet），可以获得最新的值（newValue）；另一个是属性已经发生变化（didSet），此时还可以知道变更之前的旧值（oldValue）。具体选用哪

一种，实际区别并不很大。这里我们使用 willSet，完整代码如下。

```
    varfavorite= false {
    willSet {
    if newValue  {
    favBtn.setImage( imageLiteral(resourceName: "fav"), for: .normal)
              } else {
    favBtn.setImage( imageLiteral(resourceName: "unfav"), for: .normal)
              }
           }
       }
```

因为 favorite 默认就是 false 没有被收藏的状态，它一旦发生变化，一定是变成 true，所以要把图片换成实心的 fav，否则还是换回空心的 unfav。

此时运行一下，发现单击该按钮后一段时间才有反应，这是怎么回事？把 Button 单击交互时的高亮属性（Shows Touch On Highlight）勾选即可，如图 22-68 所示。

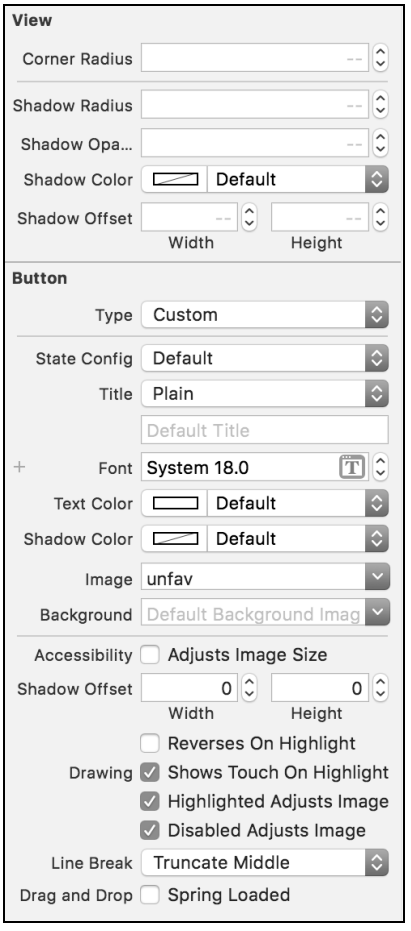


图 22-68

运行就可以单击该按钮了，单击第一个爱心按钮进行收藏。往下拖动，好像有问题，怎么下面一个爱心图片是实心的？

3. 爱心图片漏洞的处理

是否记得，单元格显示什么，都是由数据源代理的。

cellForRow 复用单元格方法实现的。我们发现这里没有更新爱心按钮图片状态的相关代码，加上以下代码：

```
cell.favorite = favorites[indexPath.row]
```

再次运行，一切完美！

总结：本节还是很有挑战性的。

作业：尝试在单元格上再加一个按钮并运行交互。

22.7　单元格滑动菜单

22.6 节我们讲的是单元格内部的 UI 组件的互动处理。其实系统也提供单元格整体互动的机制，如常见的左右滑动菜单，就可以很轻易地实现。再如在微信或 QQ 的聊天记录列表上，向右滑动其中的一条，会有一个删除的选项出现，可以把该行记录删除，这是一个非常实用的交互。除了常规的删除，还可以把聊天置顶显示等功能，下面我们就来着手实现。效果如图 22-69 所示。

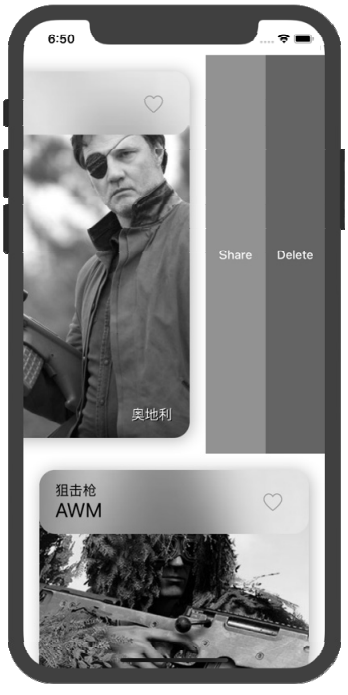


图 22-69

单元格左侧滑动菜单如图 22-70 所示。



图 22-70

1. 删除一行

按照 MVC 模式，V 是删除按钮，M 是该行对应的数据，C 是显示删除按钮，然后删除该行对应的数据。Table View 的数据源协议里提供了这个方法。读者可能要问，这个方法如何去找？很简单，单击 XCode 帮助菜单的“Developer Documentation”选项，如图 22-71 所示，即可打开苹果在线的开发者文档。

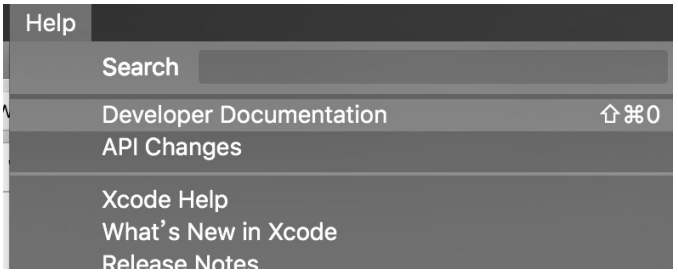


图 22-71

输入 delete table，然后搜索，在提示中就会出现匹配的文档，第一项如图 22-72 所示。

删除一行的代理方法如图 22-73 所示。

原来删除一行，只要实现含参数 commit…forRowAt 这个方法就可以了。这个代理方法位于 UITableViewDataSource 协议中，信息详细可以向下拉该文档，发现这个协议中还有很

多其他方法，读者有时间其实都可以研究一下。在 WeaponsTableViewController.swift 中，输入该方法，实现的完整代码如下。

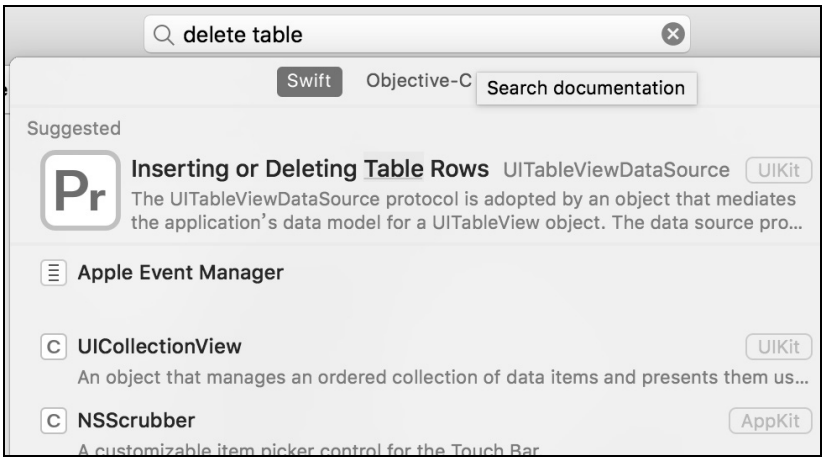


图 22-72

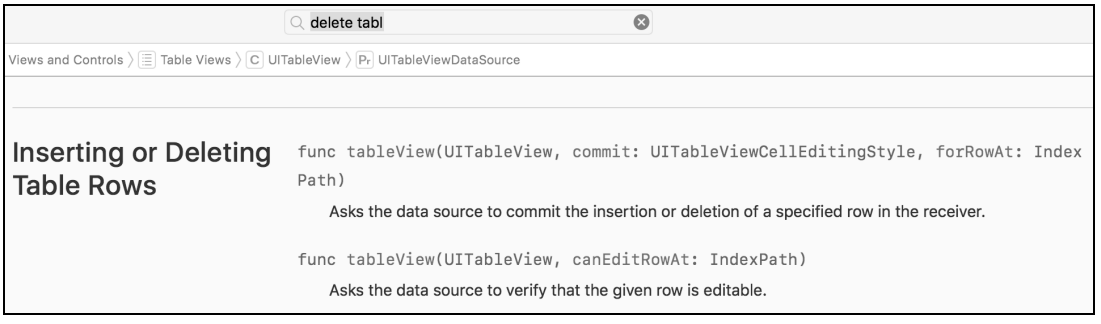


图 22-73

```
    overridefunc tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
    weapons.remove(at: indexPath.row)
    weaponTypes.remove(at: indexPath.row)
    origins.remove(at: indexPath.row)
    weaponImages.remove(at: indexPath.row)
    favorites.remove(at: indexPath.row)
          print(weapons.count,weaponTypes.count,origins.count,weaponImages.
count,favorites.count)

          tableView.deleteRows(at: [indexPath], with: .automatic)
       }
```

上述代码中的前 5 句是删除数据源中对应的数据，并在控制台显示删除后各数组中的大小，以做数据的验证。最后一句是从 Table View 中删除所在的行，这样就保证了数据（M）

和视图（V）是同步的。运行效果如图 22-74 所示。



图 22-74

2. 定制滑动菜单

仅一个删除按钮可能并不能满足要求，如还想加入一个分享按钮，可以把当前武器的介绍和图片分享给 QQ、iMessage、微信、微博好友等。

在 iOS 11 中要实现自定义的滑动菜单，需要实现以下两个方法。

```
tableView(_:leadingSwipeActionsConfigurationForRowAt indexPath:IndexPath)
tableView(_:trailingSwipeActionsConfigurationForRowAt indexPath:IndexPath)
```

从参数的名字可以看出，第一个是左侧滑动菜单（leading），第二个是右侧滑动菜单（trailing）。

使用的相关类有 UISwipeActionsConfiguration 和 UIContextualAction。UIContextualAction 是指菜单按钮类，用来定义按钮的标题和操作；UISwipeActionsConfiguration 是指菜单配置，

用来告诉系统有几个按钮。注意，一旦要自定义滑动菜单，就要自己实现删除按钮，之前的方法就不可用了。使用 trailing 这个方法，先实例化删除按钮类，然后把删除代码复制过来。

```
    overridefunc tableView(_ tableView: UITableView, trailingSwipeActions
ConfigurationForRowAt indexPath: IndexPath) ->UISwipeActionsConfiguration? {
    letdelAction = UIContextualAction(style: .destructive, title: "Delete") { (_,
_, completion) in
    self.weapons.remove(at: indexPath.row)
    self.weaponTypes.remove(at: indexPath.row)
    self.origins.remove(at: indexPath.row)
    self.weaponImages.remove(at: indexPath.row)
    self.favorites.remove(at: indexPath.row)

            tableView.deleteRows(at: [indexPath], with: .automatic)

            completion(true)
        }

    }
```

因为在闭包中引用变量，数据源要加上 self 前缀，也会自动出现代码提示。Print 这段代码也就不需要了。completion（true）参数是指操作结束，让右侧菜单消失。

接下来就是分享按钮了，按钮颜色为橙色。要分享的是武器图片和文字介绍，用 text 和 image 两个变量来保存，UIActivityViewController 是系统提供的复制、分享类，有标准的外观，凡是大型 App，都保存在这个系统标准接口中，我们只要轻松调用就好了，根本不必针对某个 App 做专门开发，省时省力。完整代码如下。

```
    let shareAction = UIContextualAction(style: .normal, title: "Share") { (_,
_, completion) in
    let text = "这是绝地求生中的帅气\(self.weapons[indexPath.row]!"
    let image = UIImage(named: self.weaponImages[indexPath.row])!

    let ac = UIActivityViewController(activityItems: [text, image], application
Activities: nil)
    self.present(ac, animated: true)
            completion(true)
        }
    shareAction.backgroundColor = .orange
```

最后根据方法返回值的要求，把按钮配置返回。在 iPhone 模拟器和 iPhone 机上运行，效果非常好，如图 22-75 所示。

iPhone 手机分享到 QQ 效果如图 22-76 所示。



图 22-75



图 22-76

iPhone 手机分享到 QQ 邮箱效果如图 22-77 所示。

iPhone 手机分享到“信息”效果如图 22-78 所示。



图 22-77



图 22-78

看起来效果不错。不过如果在 iPad 上运行，会有些小问题，因为 UIActivity ViewController 在 iPad 和 iPhone 上不同，它并不是从底部弹出的，而是以气泡弹窗形式显示的，因为 iPad 屏幕较大，需要手工给定其源视图的引用和大小。在 present 前加入以下代码：

```
    iflet pc = ac.popoverPresentationController {
    let cell = tableView.cellForRow(at: indexPath)!
                pc.sourceView = cell
                pc.sourceRect = cell.bounds
    }
```

iPad 上的弹窗分享效果如图 22-79 所示。

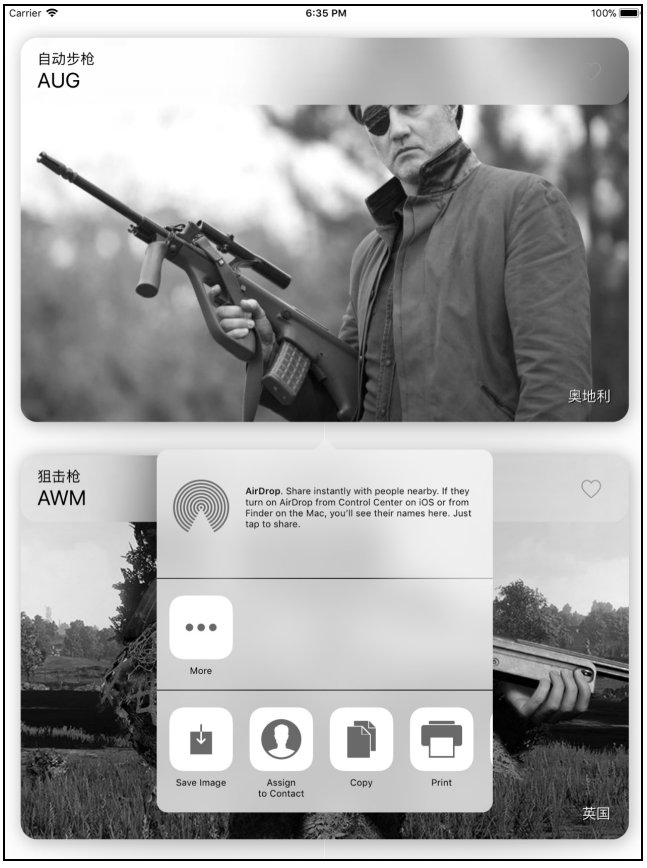


图 22-79

最后是左侧滑动菜单的实现，大同小异，收藏按钮并没有添加任何功能，完整代码如下。

```
    overridefunc tableView(_ tableView: UITableView, leadingSwipeActions
ConfigurationForRowAt indexPath: IndexPath) ->UISwipe ActionsConfiguration? {
    let favAction = UIContextualAction(style: .normal, title: "Favorite") { (_,
```

```
_, completion) in
                completion(true)
            }

            favAction.image =  imageLiteral(resourceName: "fav")
            favAction.backgroundColor = .orange

    let config = UISwipeActionsConfiguration(actions: [favAction])
    return config
        }
```

作业：实现左侧滑动菜单的收藏功能。

22.8 导航和转场

目前，我们的 App 只有一个场景，显得单调。我们很想看看武器的性能介绍、使用场景、大幅图片等。下面我们新增一个简单的场景，并用导航在两个场景中直接进行转场，最后学会在转场时传递数据。主场景和新场景的效果如图 22-80 和图 22-81 所示。

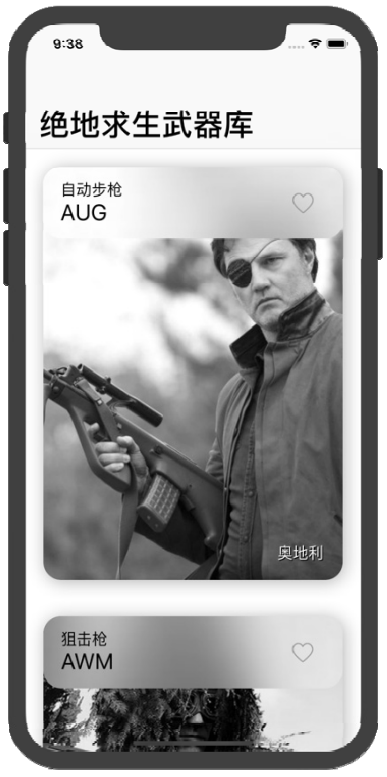


图 22-80



图 22-81

1. 导航

使用 App 免不了在各场景间进行切换，为了避免用户迷路，需要一个好的“向导”来进行控制，iOS 的这个“向导”称为导航。

我们一再提到 MVC 模式，那么导航也不能逃脱这个模式。下面举一个每天用 iOS 的例子，如我们想更改系统显示的温度单位，要进行 3 步导航才能到达设置场景如图 22-82 至图 22-84 所示。



图 22-82



图 22-83

到达温度单位场景，如图 22-85 所示。

这次导航有 4 个场景，设置首页是导航的根场景（RootViewController），“温度单位”是当前用户能看到的场景，即最上面的场景（TopViewController），这就是导航的 Model。

View 就是用户能看到的“温度单位”导航条区域（NavigationBar），表明用户当前的位置，左边还有一个返回按钮（Back Button）。导航条样式、左侧/右侧按钮、返回按钮图片都可以自定义。

Controller 是指如何控制各场景的压入（Push）和弹出（Pop），动画效果为从右向左，有返回箭头。

在各个场景间转换，称为转场（Segue）。可能读者已经看出来了，导航一般在比较有层级关系的场景中使用。

在没有层级关系的场景间转场时，使用展示（Present）和退场（Dismiss），动画效果为从下向上，没有返回箭头。



图 22-84



图 22-85

给场景加入导航非常简单，选中主场景，执行 XCode 菜单的 Editor→Embed in→Navigation Controller，如图 22-86 所示。

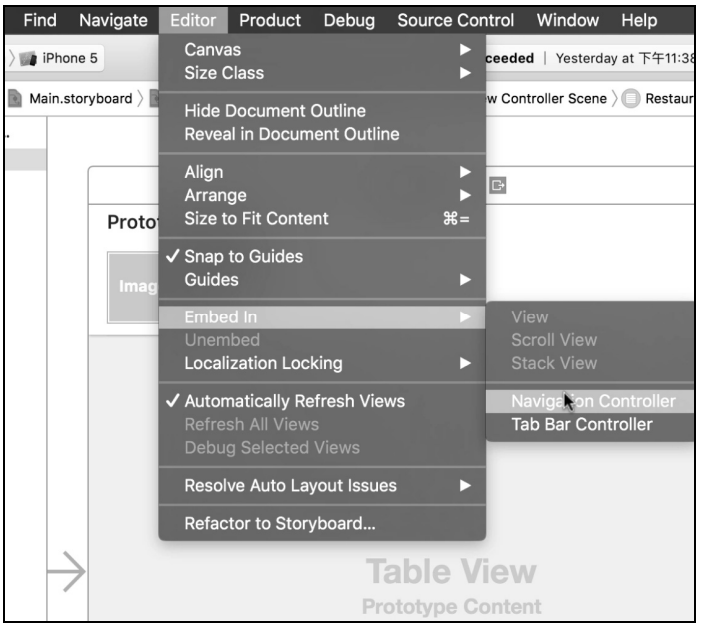


图 22-86

这时在 Storyboard 中会加入一个导航控制器，它和主场景间有一个箭头相连。

打开场景结构栏，会发现导航控制器下的 Relationship 指向主场景，即导航的根场景是主场景，如图 22-87 所示。

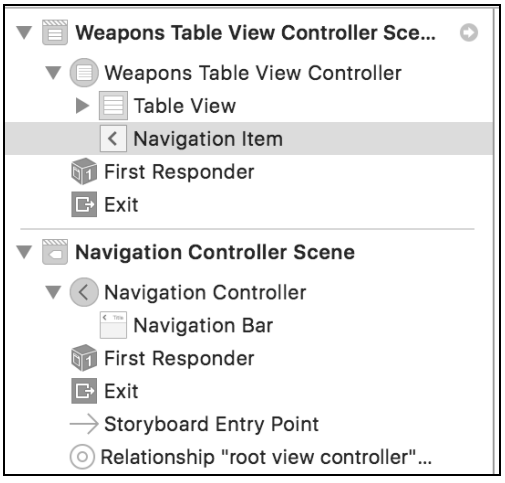


图 22-87

主场景中有一个 Navigation Item，可以在属性中设置 Title 指定主场景的导航标题，如图 22-88 所示。

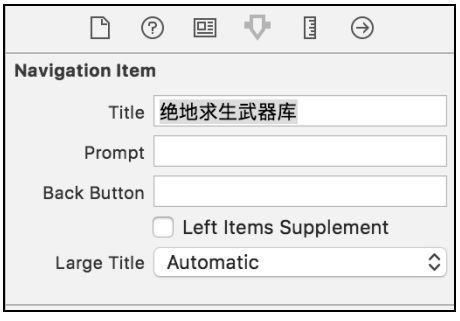


图 22-88

选择导航控制器下的 Navigation Bar，可以更改导航条的样式，勾选 Prefers Large Titles 复选项，如图 22-89 所示。

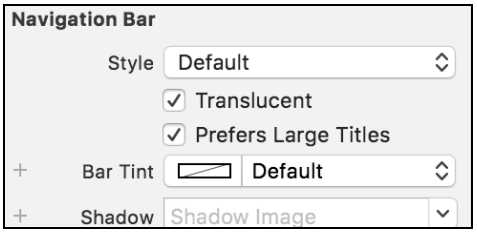


图 22-89

预览效果如图 22-90 所示。

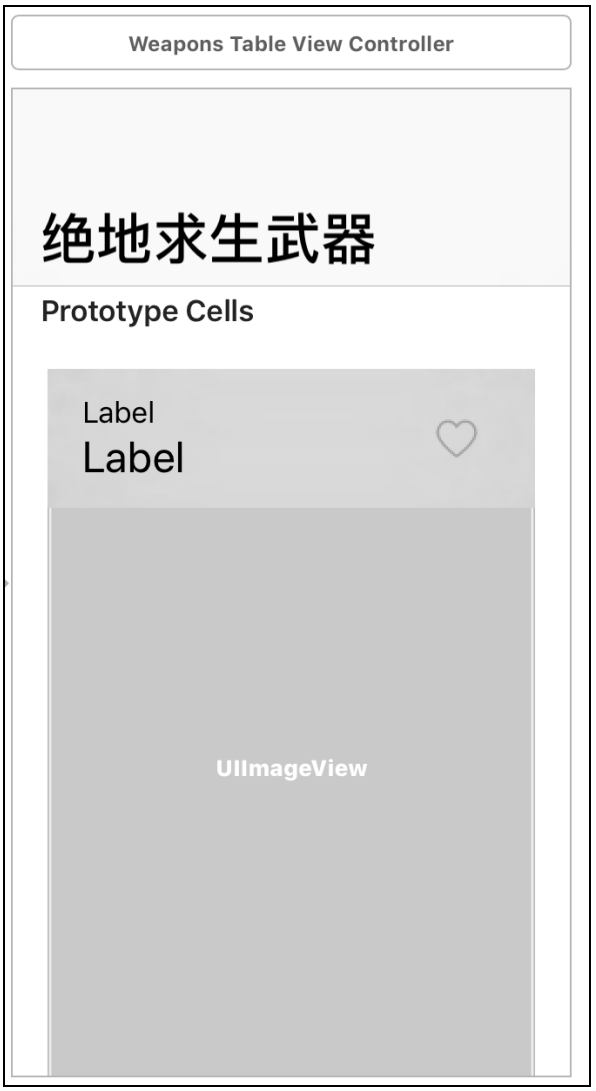


图 22-90

2. 新场景

现在新建这个代表武器详情的场景，从组件库中拖曳一个 ViewController，只拖曳一个 ImageView 位于其上，约束距离上、下、左、右为 0。填充模式为 AspectFill。效果如图 22-91 所示。

接着建立两个场景间的转场，打开场景结构栏，选择主场景的 CardCell，按住【Ctrl】键拖入新场景，如图 22-92 所示。

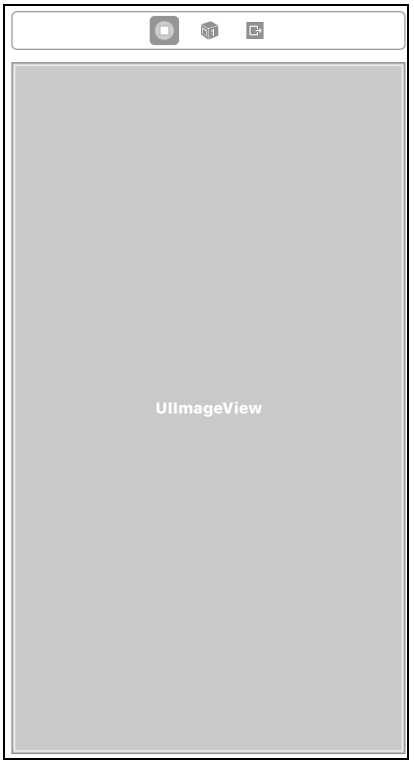


图 22-91



图 22-92

转场类型 Show 如图 22-93 所示。

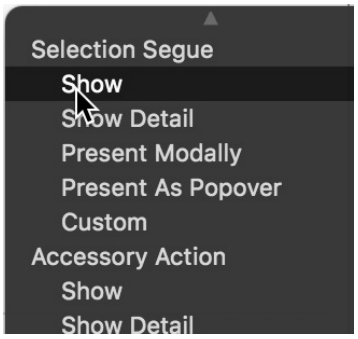


图 22-93

定义了这个转场后，场景结构中会增加 Showsegueto “ViewController”，选中它，在 Storyboard 中即可看到它的起始和目的场景，属性栏中的 Kind 是转场类型，也就是在菜单中选中的 Show。

现在可以运行看下效果，单击单元格。完美实现转场和返回。但没有新场景的图片。下面着手解决这个问题。

转场类型的详细介绍如表 22-7 所示。

表 22-7

类　型	含　义
Show/Show Detail	显示下一页，配合导航使用 相当于导航的压入场景（push） 系统自动提供返回按钮
Present Modally/ as Popover	模态展示，不需要导航的情况下 Popover 指 iPad 上的弹窗显示风格

场景 UI 和交互已经有了，接下来设置控制器。在工程文件任选一个文件执行右键菜单 NewFile→CocoaTouchClass，命名为 WeaponDetailViewController，继承自 UIViewController，保存后，与新场景连接。

因为要传递图片，所以把 ImageView 与控制器相连：

```
@IBOutletweakvar headerImageView: UIImageView!
```

再加一个属性用于传递图片名称：

```
var imageName = ""
```

3. 使用转场来传递值

转场是在两个场景直接进行切换，其中不仅是画面的转换，同时还可以进行正向、反向的数据传递。

这里我们需要从主场景进入下一级场景，即正向转场。有两个术语，源场景和目的场景，这里的源场景就是 WeaponsTableViewController.swift，目的场景是 WeaponDetail ViewController.swift。我们现在希望从正向传递数据，到源场景代码中，找到源代码下方，把 MARK：-Navigation 这段代码后面的 prepare 代码解除注释，这段代码在转场前会被执

行，这样就可以在其中传递数据了。完整代码如下：

```
    overridefunc prepare(for segue: UIStoryboardSegue, sender: Any?) {
    let row = tableView.indexPathForSelectedRow!.row

    let destination = segue.destinationas! WeaponDetailViewController
           destination.imageName = weaponImages[row]
        }
```

第一句代码是获取当前单元格所在的行。

第二句从方法的 segue 参数中，获取转场目的场景控制器，强制转换成详情控制器，因为这里只有一种转场，所以目标场景必然是详情控制器。

获取目的场景控制器后，即可给它传递当前单元格中的图片名称。

现在运行，还是没有图片

数据虽然传递了，但并没有让 ImageView 显示，在 viewDidLoad 中加入显示图片的代码：

```
    overridefunc viewDidLoad() {
    super.viewDidLoad()

    headerImageView.image = UIImage(named: imageName)
        }
```

再次运行，效果如图 22-94 所示。



图 22-94

内页其实不需要大标题，加入以下代码到 viewDidLoad 中：

```
    overridefunc viewDidLoad() {
    super.viewDidLoad()

    headerImageView.image = UIImage(named: imageName)
    navigationItem.largeTitleDisplayMode = .never
        }
```

效果如图 22-95 所示。



图 22-95

作业：加上其他字段，并做合适的布局。

第 23 章
中　级

初级我们完成了一个场景及漂亮的 Table View 布局，并讲解了 MVC。中级要更上一层楼。

23.1　Model 重构

首先要让 Model 更上一层楼，读者可能已经发现，现在有几个数组来分别保存数据，显得有些混乱。首先我们使用面向对象的方法，把这些离散的武器属性组合起来。面向对象简单来说，就是把变量和函数“归类”，从而可以像生物一样实现“进化”，这种思路符合人的常规思维，比较容易理解。

例如，以人类的进化来举例，山顶洞人的特征和行为都相当原始，如表 23-1 所示。

表 23-1

山顶洞人	
特　征	行　为
体重、肤色、年龄、性别	跑动、狩猎、进食

而现代人很显然有更丰富的特征和行为，但依然可以保持山顶洞人的一些原始特征和行为，如表 23-2 所示。

表 23-2

现代人	
特　征	行　为
国籍、职业、发型、薪水	坐车、买菜、吃饭、娱乐、学习

这样一种分类的方法，让生物学家更容易研究。编程界借鉴了这种方法，至今在所有高级编程语言中都有实现。

Swift 中的面向对象方法非常容易使用，除 iOS 系统既有框架在可预见的未来继续使用 class 外，Swift 建议使用语言本身的 struct，也就是结构体来实现面向对象，其优点是默认实现构造方法及基本上不需要关心内存管理。如果读者对 class 和 struct 还是不理解，建议去看一下 Swift 语法部分的内容，这里不再赘述。

目前主场景 Table View 用的数据源数组有几个，如图 23-1 所示，彼此之间的联系非常弱，显得零散和无凝聚力：



图 23-1

如果按面向对象的方法来组织，则看起来清晰很多，如图 23-2 所示。



图 23-2

前面讲述了 Swift 推荐使用结构体，为了清晰，我们把模型单独放到一个源码文件中。新建 Swift 文件 Weapon.swift，如图 23-2 所示编写如下对应的结构体定义。

```
struct Weapon{
var name=""
var type=""
var origin=""
var image=""
var favorite=false

//子弹类型
var bullet=0.0
//射击速度
var mv=0
//武器整体介绍
vartext=""
}
```

以上结构体定义在原来的基础上又增加了 3 个属性。现在可以对几个数组进行归一操作了。数组中的元素都是 Weapon 的一个实例。

```
    Weapon(name:"UMP9",type:" 冲 锋 枪 ",origin:" 美 国 ",image:"ump9",favorite:
false,bullet:7.62,mv:750,text:"""
```

UMP9 冲锋枪配件非常丰富，可以安装倍镜，9 毫米的子弹也比较常见。河马经常副武器选择它，因为冲锋枪的消声器四排还是比较好拿到的，切成全自动满配压枪也很稳，还可以装各种倍镜。河马本人最喜欢在上面安装全息，感觉扫射枪口稳定。河马觉得论长相，UMP9 是最符合审美的冲锋枪，所以经常拿。

（1）UMP 是一把界于冲锋枪和自动步枪之间的武器，拥有有限的后坐力和略慢的射速，同时有一个比 Vector 冲锋枪更好的有效距离（30～40 米）。

（2）UMP 需要非常多的配件来发挥全部潜力。

（3）在自动模式下开火可以迅速把人击倒，同时它使用的 9 毫米子弹非常丰富、极易找到。

完整的数组如下。

```
    var weapons=[Weapon(name:"AUG",type:"自动步枪",origin:"奥地利",image:"aug",
favorite:false,bullet:7.62,mv:900,text:"""
```

（1）AUG 作为一把自动步枪，所需的配件和 Scar 冲锋枪一样，可以装瞄准镜、枪口、握把、弹夹，使用 5.56 毫米的子弹。

（2）有人在拿到这把枪开火后直接惊呼，它的后坐力太小了吧，几乎和没有后坐力一样。

（3）和空投中的另一把自动步枪 Groza 相比，AUG 的优势也很明显，它能够装配枪口补偿器和握把，而 Groza 只能装一个消声器，这就让 AUG 的稳定性和后坐力完胜 Groza。

（4）AUG 使用 5.56 毫米的子弹，射速更快。

（5）因此在中远距离的扫射、点射中，AUG 可以完胜 Gorza，而 Gorza 仅在脸贴脸的近身战斗中更有优势，毕竟 7.62 毫米的子弹伤害力更高。

```
    """) ,
    Weapon(name: "AWM", type: "狙击枪", origin: "英国", image: "awm", favorite:
false, bullet: 0.3, mv: 910, text: """
```

AWM 狙击枪拥有游戏中最长的射程、最高的伤害力及最佳的精度（可以理解为弹道或在这个游戏里解释为归零点）。弹药类型：.300 马格楠（子弹只有空投里和枪一起捡才有）。弹匣容量：5。最大可扩展弹匣容量：10。来源：只有从空降物品中获得。可添加配件：消声器（狙击步枪）、托腮板（狙击步枪）、Extended Mag、枪口抑制器（狙击步枪）、消焰器（狙击步枪）、红点、全息、4 倍镜、8 倍镜、15 倍镜。

```
    """) ,
    Weapon(name: "十字弩", type: "冷兵器", origin: "中国", image: "crossbow",
favorite: false, bullet: 0.0, mv: 160, text: """
```

十字弩的装弹速度很慢（5 秒），在配备箭袋的情况下会快 30%左右，射击声音非常小。很难击中超过 100 米以上的物体（虽然可以装备瞄准镜），箭的下坠速度比步枪的子弹快，

不过总是可以爆头击杀裸体之人。如果只是打中对方的身体或完全没有击中对方，那么他们可能会寻找掩体，这时我们就很难击杀了。除前期可以出奇制胜外，后期不论是射程还是装弹速度上面的问题，都不推荐玩家使用这把武器。

```
    """) ,
    Weapon(name: "DP28", type: "机枪", origin: "苏联", image: "dp28", favorite:
false, bullet: 5.56, mv: 700, text: """
```

DP28 机枪非常稀有，与 98K 步枪持平（事实上，在 12 把测试服游戏中，捡到了 4 把 98K，但只看到了两把 DP28），同时该武器无法携带任何配件以提升其弹容量和稳定性，站立射击时抖动极大，具有和 AK 步枪比肩的弹道。

该武器使用高伤害力的 7.62 毫米的子弹，自带 47 发超大容量弹盘，弹道平稳，人趴下后会弹出支架，射击后坐力减少 65%以上，左右抖动明显优化，其稳定性接近满配的 Scar 冲锋枪。由于其射速较慢，稍微压枪便能造成 1000 千克的输出!如果能掌控好开枪节奏，甚至可以当作一把连狙使用——47 发不要配件的 SKS 半自动步枪。

```
    """) ,
    Weapon(name: "Groza", type: "自动步枪", origin: "俄罗斯", image: "groza",
favorite: false, bullet: 7.62, mv: 715, text: """
```

Groza 作为游戏中最新的突击步枪，装备 7.62 毫米的子弹，这也就意味着这把抢比 M16A4、M416 及 SCAR-L 三兄弟的伤害力高，与 AKM 的伤害力不分伯仲，并且这把枪只有在空投中产出。优点：中、近距离非常强，装上配件的话，射速快到在一瞬间杀掉一个人。

缺点：空投出品，比较稀少，远距离乏力。

```
    """) ,
    Weapon(name: "Kar98k", type: "狙击枪", origin: "德国", image: "kar98k",
favorite: false, bullet: 7.62, mv: 760, text: """
```

Kar 98K 又称毛瑟 98K，是以 ka98a 和 ka98b 型步枪为原型进行改装的卡宾枪型（K 为“Kurz”的缩写，德文意为“短”）。后来该枪进行了设计改进，被认为是该时代螺栓制动设计的巅峰之作。该枪自从被推出以来，已被普遍地复制在每个螺栓制动步枪中。

这是绝地求生在地图上能获取的狙击步枪，有时也存在于空投中，而其他的狙击步枪只有在空投中有。正因如此，98K 通常是一个很好的武器，可以无须冒着交战的风险去抢空投。

枪口类：狙击枪枪口补偿器、狙击枪消焰器、狙击枪消声器。

瞄准镜：红点瞄准镜、全息瞄准镜、2 倍镜、4 倍镜、8 倍镜、15 倍镜（空投）。

枪托类：98K 子弹袋、狙击枪托腮板。

```
    """) ,
    Weapon(name: "M16A4", type: "自动步枪", origin: "美国", image: "m16a4",
favorite: false, bullet: 5.56, mv: 900, text: """
```

M16A4——这个游戏有一个严重的错误，游戏里 M16 的模型是早期的装有可拆卸提把的 M16，并不是 M16A4。

真实的 M16A4 枪是使用 MIL-STD-1913 皮卡汀尼导轨和点式机械瞄具的，前护木是有下导轨的，可以装配各种提把和下挂配件。

游戏中的 M16A4 枪总体来说在步枪里的实战用途是最佳的。

并不是说 M16A4 枪的面板数据有多好，但这把枪的一切都优先为了突击步枪最基本的职责服务——中距离单发点射！

该枪面板最高的单发稳定性，以及最远的射程、优秀的后坐力可控性，在中距离对射的情况下完全碾压其他突击步枪。

M16A4 枪的其他方面都不突出，尤其是只能切换单发/三连发，但在实战中却是最实用的，因为这把枪完全只忠于它身为突击步枪该干的事，M16A4 枪是最完美的主武器选择，我们完全可以在第二武器栏装备一把霰弹枪或冲锋枪来应付近战，而将中远距离的火力完全交给 M16A4 枪。

```
    """) ,
    Weapon(name: "Micro UZI", type: "冲锋枪", origin: "以色列", image: "microuzi",
favorite: false, bullet: 9.0, mv: 350, text: """
```

Micro UZI 冲锋枪结构紧凑、动作可靠、勤务性好，且造价低，使用 9 毫米巴拉贝鲁姆手枪弹。它有木托和折叠托两种型号，木托为早期产品，折叠托为标准型。采用相同结构的变型枪有：UZI9mm 轻型冲锋枪、UZI9mm 微型冲锋枪、UZI9mm 半自动卡宾枪和 UZI9mm 手枪。它在 20 世纪 50 年代初定型，由以色列军事工业公司（IMI）生产，并作为以色列军队的制式冲锋枪，德国和比利时等国的军队也装备此枪，现已成为西方国家广泛使用的一种冲锋枪。

1948 年，在战争中，以色列人发现他们自己缺乏一种有效的冲锋枪。为此，1949 年，由以色列陆军中尉 Uziel Gal（乌兹·盖尔）参照 Vz23 系列冲锋枪和 ZK476 式冲锋枪的结构特点，并充分考虑中东地区的沙漠环境条件，成功设计了 UZI9mm 冲锋枪。

```
    """) ,
    Weapon(name: "平底锅", type: "厨具", origin: "美国", image: "pan", favorite:
false, bullet: 0.0, mv: 0, text: """
```

平底锅能挡子弹这个机制本是设计师的一个失误，但由于非常有趣且极受玩家欢迎，设计师便将这个失误保留了下来，因为平底锅没有耐久性，也使得平底锅的防御效果胜过任何其他防具。

这也就意味着，平底锅可以完美地代替头部的防御装备。又因为平底锅没有耐久性，所以几乎可以完美地实现抵挡。

平底锅是一个近战武器，就算不用，也不会让那个格子空着。

如果放入砍刀和棍子，则都会显得有些尴尬，平底锅是性价比最好的道具了。

```
    """) ,
    Weapon(name: "SKS", type: "半自动步枪", origin: "苏联", image: "sks", favorite:
false, bullet: 7.62, mv: 800, text: """
```

SKS 半自动步枪，中远距离互点霸主，在标准情况下，伤害力与 98K 步枪相比，需要比 98K 步枪多发一枪。装上扩容弹夹 20 发的子弹碾压所有步枪，可以加装狙击补偿和托腮板，在顶配的状态下，手感、伤害、射程都可以胜过绝大多数的步枪。其缺点是无法连发，且没有配件，子弹少。该枪在有扩容弹夹的情况下可以替代步枪，单挑模式选择 98K

步枪，团队模式一般选这把枪，这把枪和 98K 步枪最大的差距是，98K 步枪对二级头目能够一枪致死，而它却不能。

下面的内容是 UMP9 冲锋枪完整的武器数组代码，演示从“部分”到“整体”（有部分内容重复），可以让读者看到完整的武器数组代码，从而检查自己代码的准确性。

作者：蒸汽机器人

链接：https://www.zhihu.com/question/62539108/answer/201682194

来源：知乎

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

```
    """) ,
    Weapon(name: "UMP9", type: "冲锋枪", origin: "美国", image: "ump9", favorite:
false, bullet: 7.62, mv: 750, text: """
```

UMP9 冲锋枪配件非常丰富，可以安装倍镜，9 毫米的子弹也比较常见。河马经常副武器选择它，因为冲锋枪的消声器四排还是比较好拿到的，切成全自动满配压枪也很稳，还可以装各种倍镜。河马本人最喜欢在上面安装全息，感觉扫射枪口稳定。河马觉得论长相，UMP9 是最符合审美的冲锋枪，所以经常拿。

（1）UMP 是一把界于冲锋枪和自动步枪之间的武器，拥有有限的后坐力和略慢的射速，同时有一个比 Vector 冲锋枪更好的有效距离（30～40 米）。

（2）UMP 需要非常多的配件来发挥全部潜力。

（3）在自动模式下开火可以迅速把人击倒，同时它使用的 9 毫米子弹非常丰富、极易找到。

```
    """)
    ]
```

感觉费了九牛二虎之力把做好的数据源转换成了新的，原数据源引用的地方也要一并做修改。以下代码的注释部分为以前的代码。例如，收藏相关代码的修改：

```
    //        self.favorites[indexPath.row] = !self.favorites[indexPath.row]
          varfavorite = self.weapons[indexPath.row].favorite
          favorite = !favorite

    //        cell.favorite = self.favorites[indexPath.row]
          cell.favorite = favorite
```

删除单元格相关的代码：

```
    self.weapons.remove(at: indexPath.row)
    //          self.weaponTypes.remove(at: indexPath.row)
    //          self.origins.remove(at: indexPath.row)
    //          self.weaponImages.remove(at: indexPath.row)
    //          self.favorites.remove(at: indexPath.row)
```

分享相关代码：

```
    //          let text = "这是绝地求生中的帅气\(self.weapons[indexPath.row])!"
    //          let image = UIImage(named: self.weaponImages[indexPath.row])!
```

```
let text = "这是绝地求生中的帅气\(self.weapons[indexPath.row].name)!"
let image = UIImage(named: self.weapons[indexPath.row].image)!
```

单元格内容相关代码：

```
let weapon = weapons[indexPath.row]

        cell.typeLabel.text = weapon.type
        cell.weaponLabel.text = weapon.name
        cell.originLabel.text = weapon.origin
        cell.backImageView.image = UIImage(named: weapon.image)
        cell.favorite = weapon.favorite
```

转场前传递图片名相关代码：

```
//        destination.imageName = weaponImages[row]
        destination.imageName = weapons[row].image
```

作业：武器还有很多其他属性，尝试继续往上加 5～6 个属性。

23.2　静态单元格

我们在初级中介绍了 Table View，使用动态单元格创建了武器的列表。到了武器详细介绍的场景，需要的单元格数是固定的，这时可以用静态单元格来实现。最终效果如图 23-3 所示。



图 23-3

1. 详情页改造

首先把原来的详情页场景删除，用 TableViewController 来替代，并把原来存在的从首页单元格到详情页的转场也重新加上，转场类型依然是 Show，Identifier 为 showWeaponDetail，如图 23-4 所示。

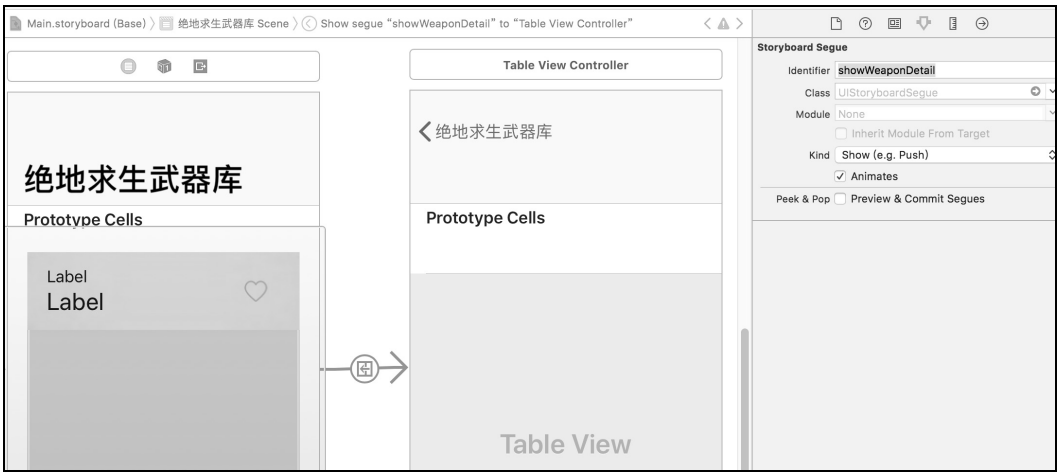


图 23-4

选中默认的原来的单元格，把 Identifier 设为 DetailCell，如图 23-5 所示。

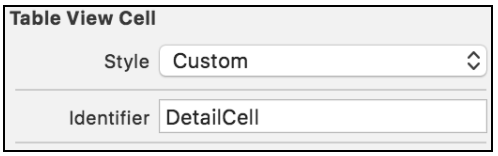


图 23-5

Table View 除了显示多行内容外，还可以在头部和尾部分别设置一个视图，作为表头和表尾，这样看起来不那么单调乏味。对于武器详情页，表头设置为一个武器的大图可能比较好看，拖曳一个 Image View 到 DetailCell 上，高度设置为 300，如图 23-6 和图 23-7 所示。

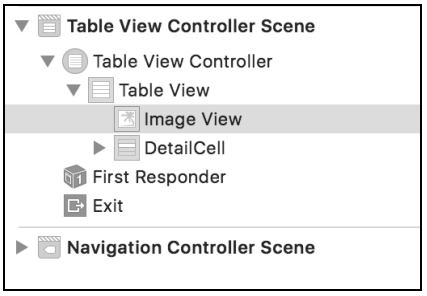


图 23-6

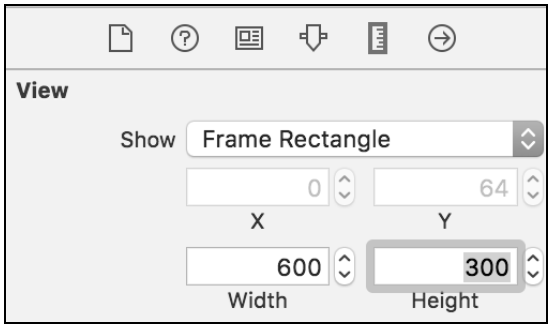


图 23-7

图像的填充模式主要有 3 种：ScaleToFill（拉伸，充满容器，但会导致图片变形）、AspectFit（居中，保持图片比例，但可能产生多余的白边）、Aspect Fill（平铺，可能会超出容器），分别如图 23-8 至图 23-10 所示。

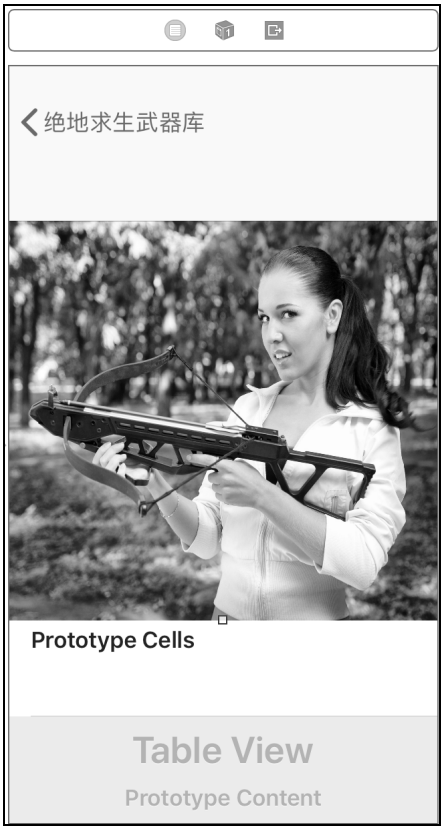


图 23-8

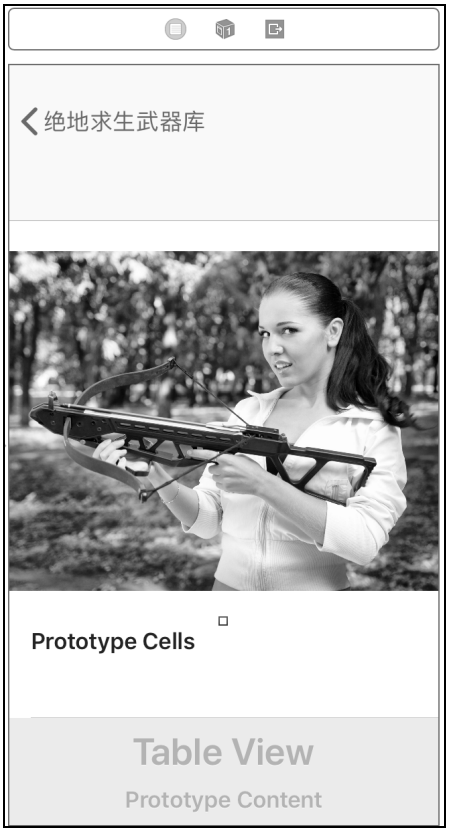


图 23-9

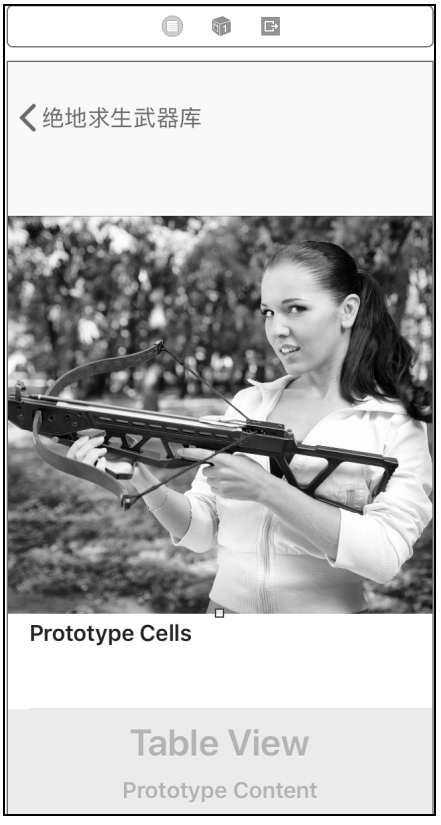


图 23-10

因为这里的图片的分辨率足够大，所以选择平铺，然后加上裁边属性，把多余的部分裁掉，也就是启用 Clip to Bounds 属性，如图 23-11 所示。

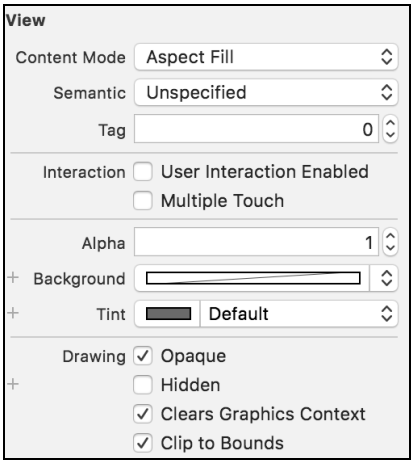


图 23-11

原型单元格中的组件和布局如表 23-3 所示。

表 23-3

组　件		约　束		
Stack View	Key （Label 亮灰）	垂直	居中	
	Value （Label）	水平	左右	15
			上下	7.5

因为 Value 的 Label 可能要显示的文字有很多行，如武器描述的内容，所以单独把它的 lines 属性设为 0，即不限制行数，如表 23-4 所示。

表 23-4

组　件	属　性	
Value （Label）	lines	0

调整 KeyLabel 的水平伸展优先级（CHP）为 252，让其在 Stack View 中更接近自身的原始尺寸，如图 23-12 所示。

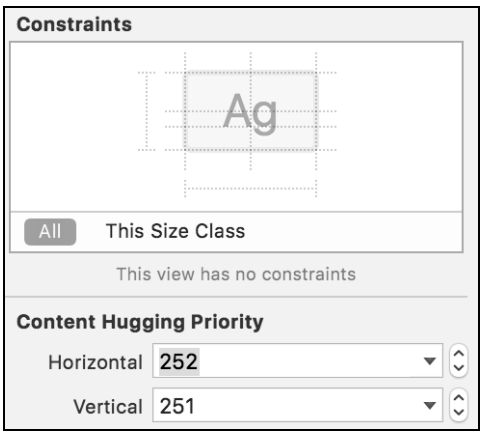


图 23-12

2. 控制器实现

View 已经布局好了，接下来就是设置 Model 和 Controller 了。给新场景添加控制器 DetailController.swift，依然是继承 UITableViewController，并在 Storyboard 上与武器详情场景关联。把 Table View 的表头，也就是 Image View，与控制器相关联：

```
@IBOutletweakvar headerImageView: UIImageView!
```

控制器对应的 Model：

```
var weapon : Weapon!
```

接着是单元格的控制器，命名与 Identifier 一致，也就是 DetailCell.swift，把两个 Label 与单元格控制器相关联：

```
@IBOutletweakvar keyLabel: UILabel!
```

```
    @IBOutletweakvar valueLabel: UILabel!
```

万事俱备，接下来就等着填充 Table View 的单元格了。这就是我们熟悉的 Table View 的数据源代理了。把默认的显示 section 的代理方法删除。

首先是显示多少行的代理方法，除了收藏与否和图片，只需要显示其他 6 个属性：

```
    overridefunc tableView(_ tableView: UITableView, numberOfRowsInSection
section: Int) ->Int {
    return6
       }
```

接着是单元格的内容，完整代码如下。

```
    overridefunc tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) ->UITableViewCell {
    let cellId = String(describing: DetailCell.self)

    let cell = tableView.dequeueReusableCell(withIdentifier: cellId, for:
indexPath) as! DetailCell

    switch indexPath.row {
    case0:
              cell.keyLabel.text = "Name"
              cell.valueLabel.text = weapon.name
    case1:
              cell.keyLabel.text = "Type"
              cell.valueLabel.text = weapon.type
    case2:
              cell.keyLabel.text = "Origin"
              cell.valueLabel.text = weapon.origin
    case3:
              cell.keyLabel.text = "Bullet"
              cell.valueLabel.text = weapon.bullet.description
    case4:
              cell.keyLabel.text = "Muzzle Veolcity"
              cell.valueLabel.text = weapon.mv.description
    default:
    cell.keyLabel.isHidden = true
              cell.valueLabel.text = weapon.text
          }

    return cell
       }
```

代码中的 cellid 是获取 Identifier，然后复用为 DetailCell，switch 部分是判断当前在哪一行，以便给单元格填充相应的属性值。

下面设置表头图片，把导航条恢复为正常的宽度：

```
overridefunc viewDidLoad() {
super.viewDidLoad()
headerImageView.image = UIImage(named: weapon.image)
navigationItem.largeTitleDisplayMode = .never
}
```

最后不要忘记修改转场代码：

```
overridefunc prepare(for segue: UIStoryboardSegue, sender: Any?) {
if segue.identifier == "showWeaponDetail" {
let row = tableView.indexPathForSelectedRow!.row

let destination = segue.destinationas! DetailController
          destination.weapon = weapons[row]
      }
   }
```

运行效果如图 23-13 所示。



图 23-13

为了使显示效果更好，武器详情页还可以做些小优化。调整 Table View 的背景亮度，如图 23-14 所示。

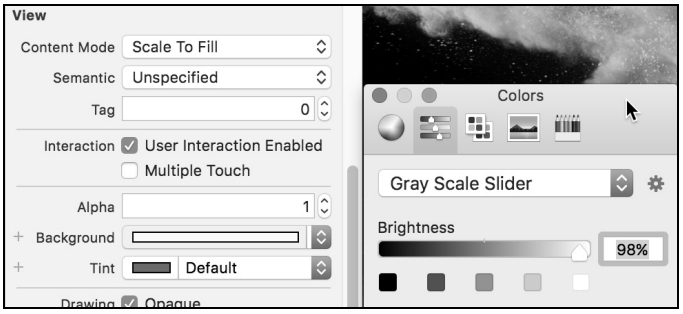


图 23-14

把单元格调整为透明色，即把 Background 属性设置为 ClearColor，Table View 的分隔线（separator）颜色调整为灰度 90%。

作业：实现一个更漂亮的场景布局。

23.3 导航栏、状态条和动态字体的定制技巧

本节介绍导航栏、状态条和动态字体的定制技巧。

1. 导航栏的定制

一般 App 都有一个主题色，如小黄车是浅黄色。我们这个 App 也定义一个主题色，这样可以到处使用，换起来也方便。定义主题色非常简单，在 Assets 中，单击左下角的“+”号，新建一个颜色集，如图 23-15 所示，Name 设置为 theme。RGB 值为 59、100、171（小波说雨燕 logo 主题色），如图 23-16 所示。

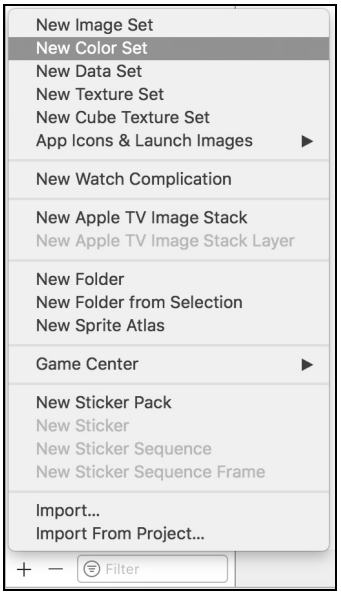


图 23-15

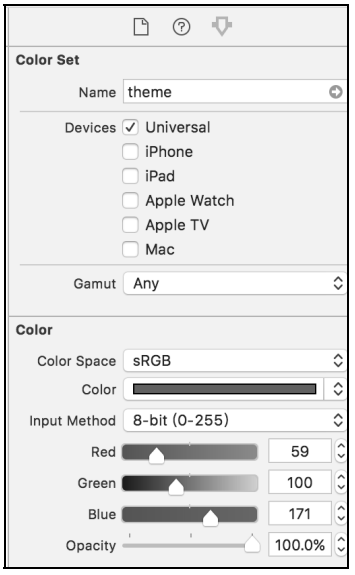


图 23-16

有了主题色以后，就可以在各种使用颜色的地方使用了。首先是导航大标题的字体颜色，选中导航控制器的 Navigation Bar，在 Larget Title Text Attributes 下的 Title Color 中选择 theme，如图 23-17 和图 23-18 所示。即刻就可以在 Storyboard 上看到效果了。

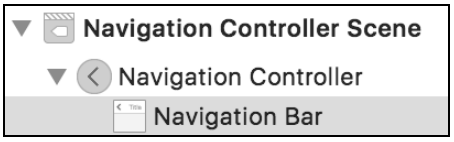


图 23-17

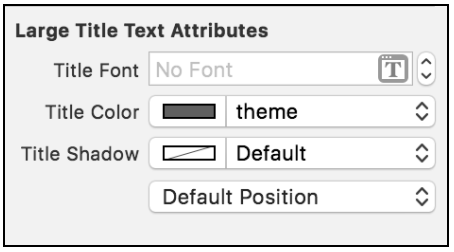


图 23-18

由于 XCode 的漏洞，运行起来可能会出错。XCode 可能会在下一个版本修复此漏洞，把颜色调回 Default。在打开 WeaponsTableViewController.swift 的 viewDidLoad 中加入以下代码：

```
navigationController?.navigationBar.largeTitleTextAttributes = [
NSAttributedStringKey.foregroundColor : UIColor(named: "theme")
        ]
```

因为详情页表头是一张图片，所以想要让导航栏透明，在 DetailController.swift 的 viewDidLoad 中加入代码：

```
navigationController?.navigationBar.setBackgroundImage(UIImage(), for:.default)
navigationController?.navigationBar.shadowImage = UIImage()
```

效果如图 23-19 所示。



图 23-19

此时导航栏已经透明了，但因为加了导航，Table View 会默认留出导航栏的位置，可以把 Content Insets 设置为 Never 即可，如图 23-20 所示。



图 23-20

调整后的效果如图 23-21 所示。



图 23-21

不过现在还是有不满意的地方，在返回按钮中有上一页的标题，这显得多余。选中上一页的 Navigation Item，在 Back Button 中输入一个空格后回车，运行后效果如图 23-22 所示。



图 23-22

默认的返回按钮比较难看，我们自定义一个。导入 back.png 到 Assets，拖到 3×部分，如图 23-23 所示。

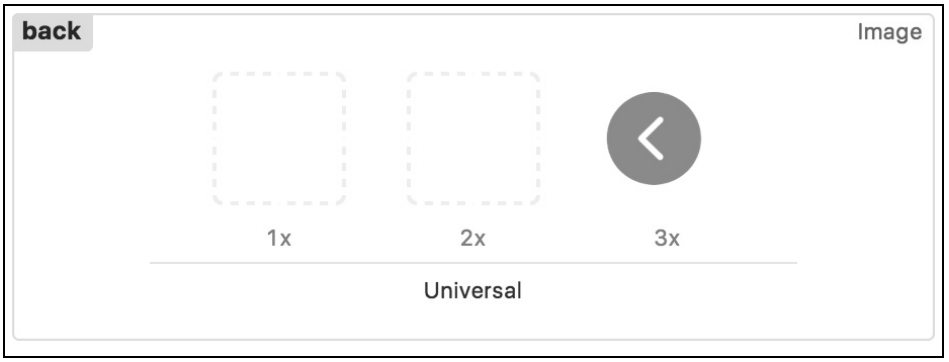


图 23-23

选中 Navigation Bar，在属性中把 Back 和 Back Mask 设置为 back，如图 23-24 所示。

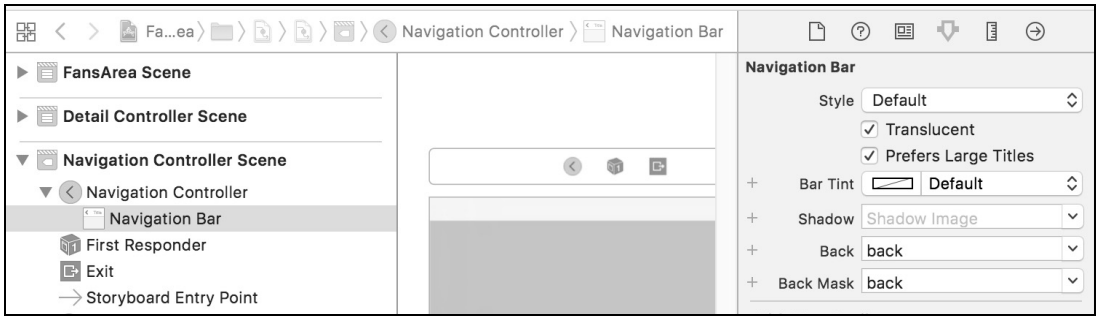


图 23-24

在 DetailController.swift 的 viewDidLoad 中加入以下代码，将导航栏字体颜色设置为主题色：

```
navigationController?.navigationBar.tintColor = UIColor(named: "theme")
```

运行效果如图 23-25 所示。

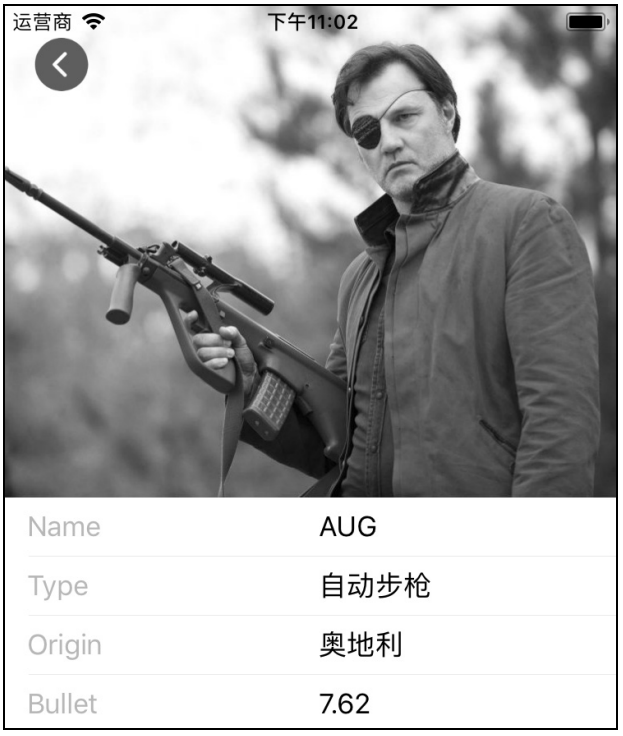


图 23-25

在滑动浏览时，返回按钮看起来比较碍事，想把它隐藏起来，浏览到顶部时返回按钮再重新出现。这时可以选择 Navigation Controller，设置属性 Hide Bars，勾选 On Swipe 复选项，如图 23-26 所示。

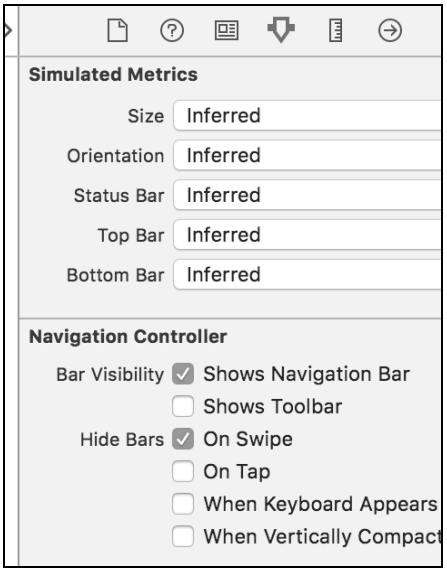


图 23-26

2. 状态条的颜色

如何设置状态条（电池和信号条）的颜色呢？首先要确定一点，状态条只有两种颜色，即黑色和白色。这里分两种情况：一种是全部页面设为一种颜色；另一种是每个页面需要不同的状态条颜色。第一种情况非常简单，只要设置导航条的样式即可。Style 默认是 White，状态条为了形成反差自动成为黑色，反之，将 Style 调成 Black，状态条就自动成为白色。

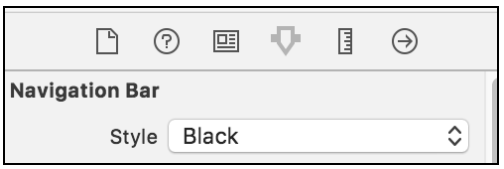


图 23-27

在多数情况下，每个页面需要不同的状态条颜色，因为每个场景的背景色有深有浅。每个场景单独设置状态条颜色很简单，如 DetailController.swift，只要重载以下属性即可。

```
    overridevar preferredStatusBarStyle: UIStatusBarStyle {
    return .lightContent
        }
```

运行后，我们发现没有生效。

因为导航控制器统一控制了导航栏的样式，默认的并不是子页面的设置。要想让子页面的状态条设置生效，必须更改这种默认设置。使用扩展来做这种操作比较方便，新建 NaviExt.swift，加入以下代码：

```
    import  UIKit

    extensionUINavigationController {
```

```
    openoverridevar childViewControllerForStatusBarStyle: UIViewController? {
    returntopViewController
        }
    }
```

childViewControllerForStatusBarStyle 属性是指状态条样式由哪个子页面提供，这里由导航最前面的场景提供。虽然属性比较长，但有个记忆技巧，只要输入 statusBar 就会有代码提示。

读者可能要问，在没有导航控制器的情况下，如何全局控制状态条颜色呢？很简单，只要在 AppDelegate.swift 的 didFinishLaunchingWithOptions 方法中加入如下代码即可。

```
    UIApplication.shared.statusBarStyle = .lightContent
```

这样运行后就正常了，效果如图 23-28 所示。



图 23-28

3. 动态字体

动态字体是 iOS 推出的一项辅助功能，方便视力欠佳的群体更容易看清楚 App 上的文字，只要在选择系统设置→通用→辅助功能→更大字体→辅助功能中的更大字体即可，如图 23-29 所示。

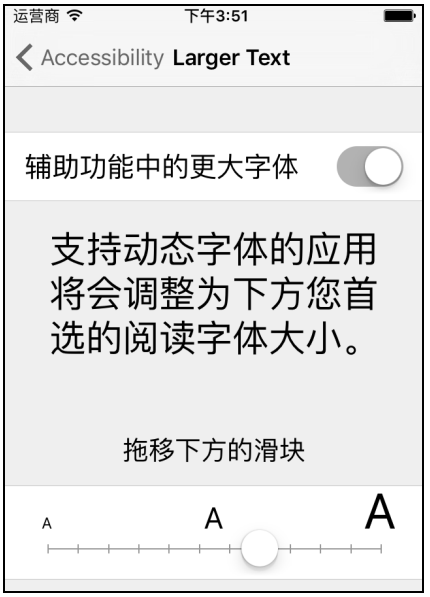


图 23-29

XCode 同时提供了一个快捷的调节功能，执行 XCode 菜单 OpenDevelop Tool→Acceesibility Inspector。

虽然动态字体功能强大，但 App 却需要主动适配，否则不会生效。适配也很简单，只要在设置字体时，选择 TextStyles 中的一种即可，并勾选 Dynamic Type 后面的 Automatically Adjusts Font 复选项，如图 23-30 和图 23-31 所示。

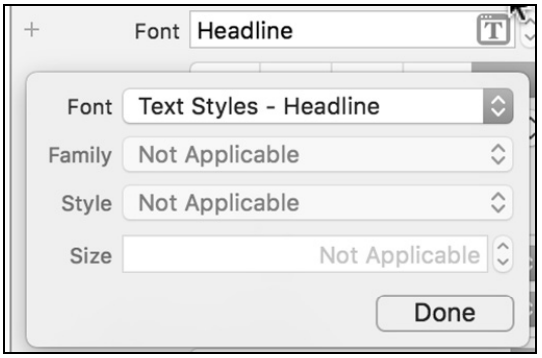


图 23-30

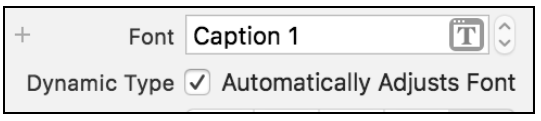


图 23-31

例如，把详情页的 Value Label 调整成动态字体，运行效果如图 23-32 所示。



图 23-32

作业：（1）对所有 Label 应用动态字体。（2）解决从详情页返回后，主页导航栏样式未恢复的问题。

23.4 视差动画

视差通常用于游戏程序中，随着用户视角的移动，给予不同物体不同的运动速度和大小变化，以营造距离和空间感。

本节要实现的方法比较简单，在用户滚动详情页时，武器图片的大小发生变化，向上滑动、正常大小和向下滑动的效果分别如图 23-33 至图 23-35 所示。



图 23-33



图 23-34



图 23-35

涉及视图的位置和移动，必须先了解一下坐标系。坐标系有绝对和相对之分，例如，我们想去距离家门口 10 米的小店买东西，或者直接说完整的地址，即某市某区某路某号。

iOS 中同样如此，一个视图的 bounds 属性是指自身坐标系中的位置和大小，frame 属性是指它在父视图坐标系中的位置和大小。

以 iPhone X 的竖屏为例，如在根视图 View 的中间有一个 200×300 点的 Image View，如图 23-36 所示。

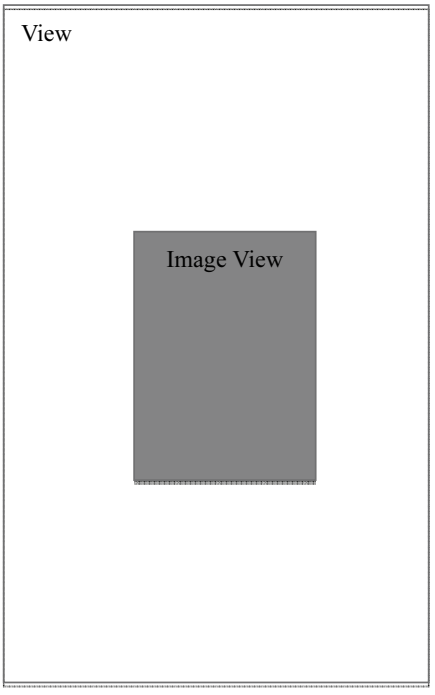


图 23-36

View 的 bounds 和 frame 属性如表 23-5 所示。

表 23-5

View		
bounds		frame
x	0.0	0.0
y	0.0	0.0
宽（点）	375.0	
高（点）	812.0	

首先要明确几个要素，宽度是与坐标系无关的。坐标（*x*，*y*）是指左上角点的位置，它们的数值是小数类型。

因为 View 本身是根视图，它没有上层的视图，所以左上角点的坐标值是相同的。

Image View 的 bounds 和 frame 属性如表 23-6 所示。

表 23-6

Image View		
bounds		frame
x	0.0	87.0
y	0.0	256.0
宽（点）	200.0	
高（点）	300.0	

Image View 在父视图 View 中居中，用简单的数学即可算出左上角点的坐标位置。

更改视图的 frame 或 bounds 的值，就会导致视图相当于父视图或自身的位置和大小变化，可以用来制作动画效果（当然制作动画还有更高级的方法）。

读者应注意到 Table View 中如果单元格比较多，会自动出现滚动条。滚动条也是一个单独的视图组件，称为 Scroll View，因为 Table View 本身是 Scroll View 的子类，所以直接使用 Scroll View 的各种方法，但 Table View 的特点是不能水平滚动。

监测滚动条发生了滚动，可以通过 UIScrollViewDelegate 中的 func scrollViewDidScroll (_scrollView: UIScrollView）方法来检验，因为详情页是 Table View Controller，默认已经包含了 UITableViewDelegate，所以无须遵从 UIScrollViewDelegate 协议。

滚动的幅值，即位移，可以通过 scrollView.contentOffset 获取，因为 Table View 中没有水平滚动，所以直接取 scrollView.contentOffset.y 即可。向上滚动，*y* 是正值；向下滚动，*y* 是负值。

现在我们想对表头的 Image View 进行视差处理，但 Table View 的特殊之处是它会自动管理表头区域，只要指定表头高度即可，其他无须操心。所以相对表头进行大小变化的操作，只能用代码把表头视图单独提取出来，占据表头所在的位置即可，成为一个“表面表头”。

首先用一个属性来保持表头视图的引用：

```
var headerView: UIView!
```

然后在 viewDidLoad 中，加入以下代码：

```
    //保留表头视图引用
    headerView = tableView.tableHeaderView
    //表头重置为 nil
    tableView.tableHeaderView = nil
    //再添加回来，变相实现了表头
    tableView.addSubview(headerView)
```

Table View 要给已经分离的表头留出位置，即顶边距：

```
    tableView.contentInset = UIEdgeInsets(top: 300, left: 0, bottom: 0, right:
0)
```

有了这个铺垫后，我们就可以对滚动进行处理了。下面我们在 DetailController.swift 中加入以下代码：

```
    overridefunc scrollViewDidScroll(_ scrollView: UIScrollView) {
    //位移
    let offsetY = scrollView.contentOffset.y
    //下移是负值
    print("纵向位移：",offsetY)
        }
```

运行打开模拟器，进入详情页，我们会发现控制器台输出“纵向位移：−300.0”，我们并没有拖动啊？这是由于设置内边距顶边为 300 的原因。

那么用户滚动后，“表面表头”的大小和位置（即 frame）相对于它的父视图 Table View 应如何变化呢？因为 Table View 有了顶边的内边距，“表面表头”就应向上移动，所以“表面表头”的纵坐标是负值；位移向上的坐标是负值，位移向下的值也是负值，所以滑动时“表面表头”纵坐标就是纵向位移：offsetY。高度始终是正值，所以是−offSetY，即加入以下代码：

```
    headerView.frame = CGRect(x: 0, y: offsetY, width: scrollView.bounds.width,
height: -offsetY)
```

要保持 Image View 大小变化而图片不失真，填充模式选择 Aspect Fill，也就是平铺，这样图片会自动根据容器大小进行调整，因此会产生视差效果。

23.5　特效、动画和手势

本节内容很丰富，包括特效、动画和手势，看看三者能擦出什么样的火花。

1. 打分场景

下面设置打分场景：入口就设在详情场景的下方，加一个“打分”按钮，单击后弹出一个背景虚化的新场景，各种打分项以动画方式出场。

因为详情页是 Table View，所以只能加一个 Footer View（即表尾），拖曳一个 Button

到原型单元格下面即可，各项设置如表 23-7 所示。

表 23-7

视　图	属　性	
Footer View	height	50
Button	title	Rate
	（View）background	主题色（theme）
	文字颜色	White

运行效果如图 23-37 所示，返回按钮和打分按钮相得益彰。

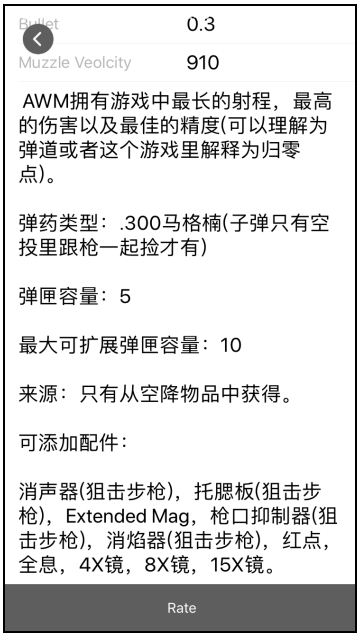


图 23-37

下面创建打分的场景。拖曳一个 View Controller 到 Storyboard 上作为点评的场景，并加一个控制器 Rate Controller.swift，点评场景添加了详情页的模糊背景效果，也就是说新场景的背景是透明的：clear color。

为了维持这种模糊效果，必须在 Modal 转场时把 presentation 设置为 Over Current Context。

转场各项设置如表 23-8 所示。

表 23-8

Segue	属　性	
Detail Controller 到 Rate Controller	Kind	Present Modally
	Presentation	Over Current Context

添加一个背景虚化组件到新场景，各项设置如表 23-9 所示。

表 23-9

背景虚化组件	约　束	
Visual Effect View	上下左右 （与 Super View）	0

为了让用户能返回上一个场景，添加一个 Tap 手势到 View，然后与控制器建立一个 IBAction，即单击后让打分场景退场，代码如下：

```
@IBActionfunc tapBackground(_ sender: UITapGestureRecognizer) {
self.dismiss(animated: true)
    }
```

虚化效果如图 23-38 所示。

下面增加打分按钮组到新场景。将 5 个按钮组合到一个 Stack View 里并居中，内部左对齐，间距为 10，把背景虚化的样式设置为 Dark，如图 23-39 所示。

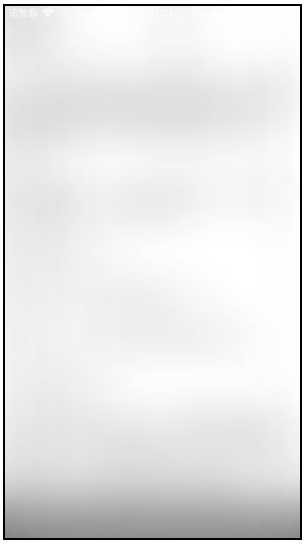


图 23-38



图 23-39

2. 动画

动画是用一组图片快速切换，来模拟动作和图形转换的过程。玩过 FPS 游戏（如绝地求生）的人都知道，每秒 30 帧才能看起来比较流畅。例如，一个 2 秒内从小变大的小球，首帧和尾帧的画面很好确定，难度在于中间帧的画面如何生成。幸运的是，在 iOS 中几乎不用考虑中间帧的动画，有很好的动画相关类来帮我们处理。

先把 UI 组件和控制器连接起来。因为这里有很多按钮，可以用 Outlet Collections: 同时让多个 UI 组件连接到同一个 Outlet 组。

找到第一个 Button，拖曳一个 Outlet Collections 到控制器中。单击此 IBOutlet 前的圆形加号按钮，依次连接其他所有的 Button。

如何分辨是哪个按钮呢？按照 Button 的连接顺序确定，如 rateButtons[0]代表第一个连接，如图 23-40 所示。

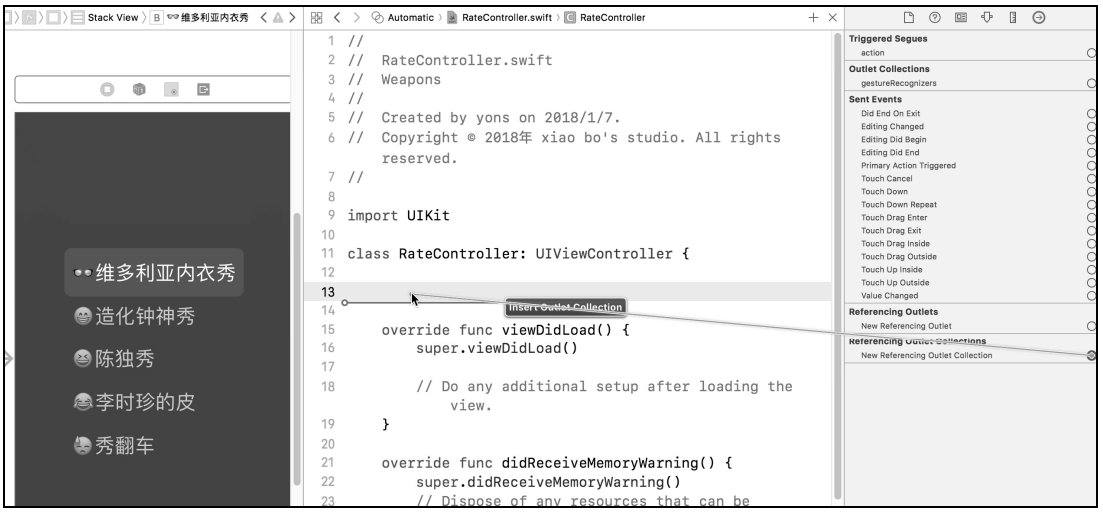


图 23-40

```
@IBOutletweakvar stackView: UIStackView!
@IBOutletvar rateBtns: [UIButton]!
```

确认一下 Outlet Connections 连接是否正确，如图 23-41 所示。

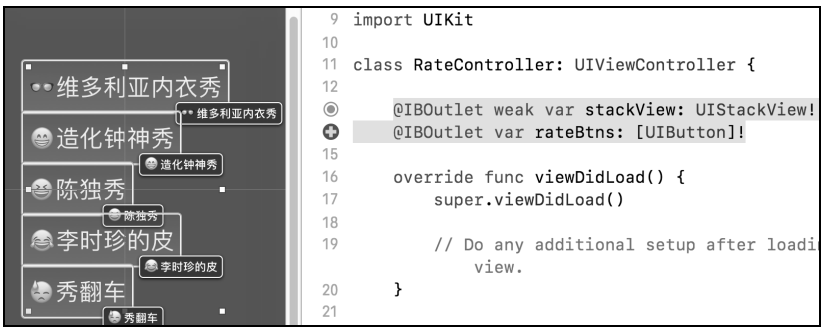


图 23-41

第一个实现 Stack View 整体的动画，设置一个从右侧滑入的效果。这里使用的方法是 UIViewPropertyAnimator 类，参数配置如表 23-10 所示。

表 23-10

动　画	**UIViewPropertyAnimator**
对象	Stack View
类型	平移
时长（秒）	0.3
动画效果	淡入

要对一个 View 的形状进行调整，有一个属性 Transform 可以使用，可使用的变换如表 23-11 所示。

表 23-11

CGAffineTransform	2D 变换
scale	缩放
rotate	旋转
translate	平移

下面确定动画的首帧、尾帧时间点及代码，在 viewDidLoad 时，把 Stack View 放到右侧屏幕之外，也就是 translation 参数所表示的，x 是 800。在 viewDidAppear 时再展示一个 0.4 秒的动画，还原 stack View 的位置，即 identity 参数。完整的说明如表 23-12 所示。

表 23-12

帧	效　果	Timing（时间点）	代　码
首帧	右侧屏幕外	viewDidLoad	stackView.transform = .init(translationx: 800, y: 0)
尾帧	复原	viewDidAppear	let animator = UIViewPropertyAnimator(duration:0.4, curve: .easeIn) { self.stackView.transform= .identity } animator.startAnimation()

运行后，动画帧截图如图 23-42 至图 23-45 所示。



图 23-42



图 23-43

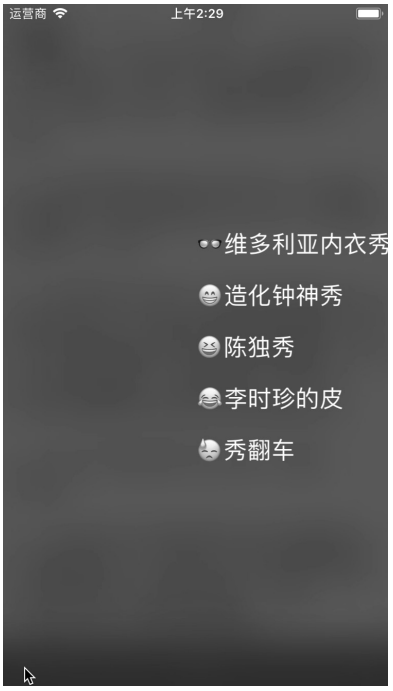


图 23-44

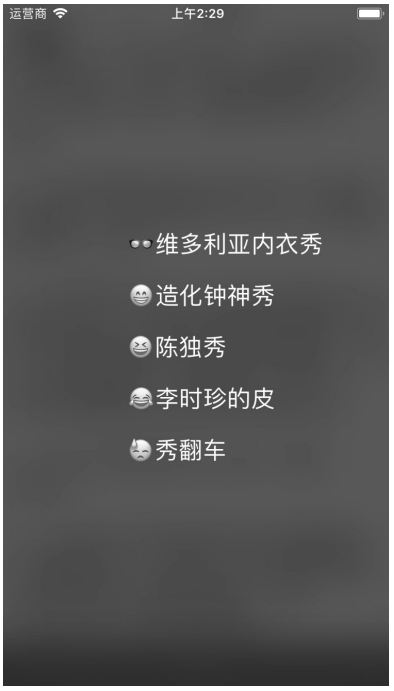


图 23-45

效果不错！不过还可以更进一步设置，让各按钮依次可见。把 stackView 相关的动画代码都删除，加入如表 23-13 所示的代码。

表 23-13

帧	效　果	Timing（时间点）	代　码
首帧	完全透明	viewDidLoad	let startPosition = CGAffineTransform(translationX: 500, y:0) for button in rateButtons{ button.alpha = 0 button.transform = startPosition }
尾帧	复原	viewDidAppear	UIViewPropertyAnimator(duration:0.4, dampingRatio:0.5) { self.rateBtns[0].alpha = 1 self.rateBtns[0].transform = .identity }.startAnimation(afterDelay:0.1)

dampingRatio 是反弹幅值，afterDelay 是延迟时间，每个按钮间的动画延迟是 0.1 秒，其余按钮的动画代码如下。

```
UIViewPropertyAnimator(duration: 0.4, dampingRatio: 0.5) {
self.rateBtns[1].alpha = 1
self.rateBtns[1].transform = .identity
```

```
        }.startAnimation(afterDelay: 0.2)

UIViewPropertyAnimator(duration: 0.4, dampingRatio: 0.5) {
self.rateBtns[2].alpha = 1
self.rateBtns[2].transform = .identity
        }.startAnimation(afterDelay: 0.3)

UIViewPropertyAnimator(duration: 0.4, dampingRatio: 0.5) {
self.rateBtns[3].alpha = 1
self.rateBtns[3].transform = .identity
        }.startAnimation(afterDelay: 0.4)

UIViewPropertyAnimator(duration: 0.4, dampingRatio: 0.5) {
self.rateBtns[4].alpha = 1
self.rateBtns[4].transform = .identity
        }.startAnimation(afterDelay: 0.5)
```

此时再运行，就可以看到按钮依次入场的效果了，如图 23-46 所示。



图 23-46

动画还可以进行组合，把 2D 变换中的平移和缩放进行组合，代码如下。

```
let startPos = CGAffineTransform(translationX: 500, y: 0)
let startScale = CGAffineTransform(scaleX: 10, y: 10)

for btn inrateBtns {
```

```
            btn.alpha = 0
            btn.transform = startPos.concatenating(startScale)
        }
```

scale 是放大的倍数。

3. 手势交互动画

结合手势和动画，可以产生与 UI 互动的高级动画效果。

可以为 View 选择不同的手势互动。手势有很多种，拖动就是其中的一种，称之为 Pan。拖动一个 Pan 手势到 stack View 上，已附加手势组件的 View，连线栏会出现手势组件的 Outlet。Pan 手势如图 23-47 所示，已附加到 stackView 上的 Pan 手势连接栏示意图如图 23-48 所示。

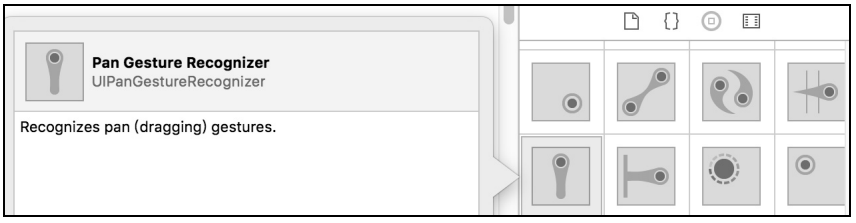


图 23-47



图 23-48

下面设置 Pan 手势的各种状态，右键拖动手势到控制器代码中，类型为 Action，命名为 dragStackView，代码如下。

```
@IBActionfunc dragStackView(_ sender:UIPanGestureRecognizer){

switch sender.state {
case.changed:
let translation=sender.translation(in: view)
let position=CGAffineTransform(translationX:translation.x,y:translation.y)
let angle = sin(translation.x / stackView.frame.width)
stackView.transform = position.rotated(by: angle)
case.ended:
UIViewPropertyAnimator(duration:0.5,dampingRatio:0.5,animations:{
self.stackView.transform=.identity
```

```
            }).startAnimation()
    default:
    break
           }
       }
```

判断手势的状态：

（1）发生拖动。

① 把手势的位置转换成在当前 View 中的位置（x，y）。

② 平移变换。

③ 旋转变换，顺着拖动方向的（x 轴）角度倾斜。

（2）拖动结束，复原位置的动画。

此时可以运行试一试效果。如果不能拖动，则勾选 stackView 的 User Interaction Enabled 复选项。

向左和向右拖动 stackView 的效果如图 23-49 和图 23-50 所示。

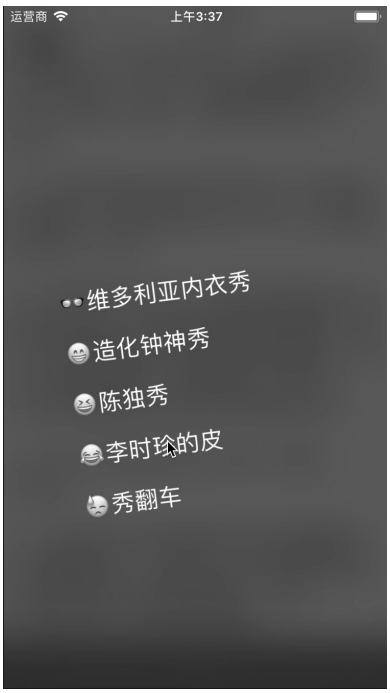


图 23-49



图 23-50

4. 反向转场传值

这是本节最后一部分内容，单击任何一个打分按钮，把分数传回上一页。这里涉及一个名词叫作反向转场。反向转场需要以下两个步骤。

（1）在上一页场景中加入一个方法，参数必须是 UIStoryboard 类型，修饰符 @IBAction，即在 DetailController.swift 中加入：

```
@IBActionfunc backToDetail(segue: UIStoryboardSegue)  {


 }
```

（2）右键拖动内页要返回的 UI 组件到 Exit 处，即可创建一个 Manual Segue，并命名为 identifier，这里是 5 个按钮，从最高评价到最低评价的 segueIdentifier 分别是 5/4/3/2/1。

右键拖动按钮到场景的 Exit 如图 23-51 所示。



图 23-51

选择反向转场方法如图 23-52 所示。

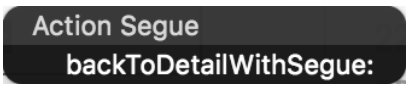


图 23-52

命名转场的 Identifier 如图 23-53 所示。

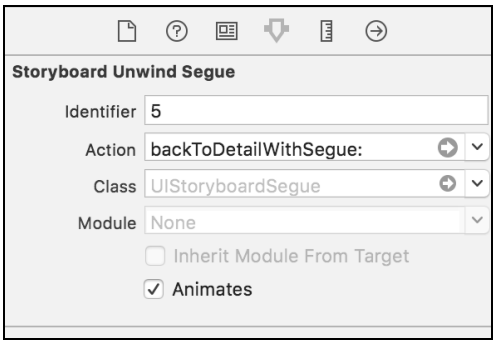


图 23-53

场景结构中的反向转场列表如图 23-54 所示。

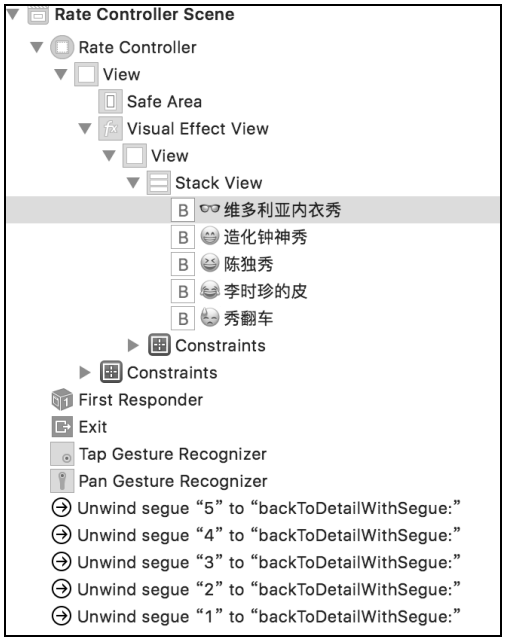


图 23-54

UI 和互动都已经配置完毕，接下来就是 Model 了，因为新增了打分属性，需要变更 Model，为了不影响已有的 Weapon 初始化方法，可增加构造器方法，完整代码如下。

```
    struct Weapon {
    var name = ""
    var type = ""
    var origin = ""
    var image = ""
    var favorite = false
    var bullet = 0.0
    var mv = 0
    var text = ""
    var rating = ""


    init(name: String, type: String, origin: String, image: String, favorite:
Bool, bullet: Double, mv: Int, text: String) {
    self.name = name
    self.type = type
    self.origin = origin
    self.image = image
    self.favorite = favorite
```

```
    self.bullet = bullet
    self.mv = mv
    self.text = text
        }

    }
```

接下来回到详情页控制器，在反向转场的代码里，可以接收回传过来的值了，更新到 Model 中，代码如下：

```
    @IBActionfunc backToDetail(segue: UIStoryboardSegue)  {
    iflet rating = segue.identifier {
    weapon.rating = rating

           }
    }
```

UI 上还没有更新打分，把打分按钮与控制器相连：

```
    @IBOutletweakvar ratingBtn: UIButton!
```

更新 backToDetail 方法的代码如下。

```
    @IBActionfunc backToDetail(segue: UIStoryboardSegue)  {
    iflet rating = segue.identifier {
    weapon.rating = rating
    ratingBtn.setTitle("打分：\(weapon.rating)星", for: .normal)
           }
        }
```

打分效果如图 23-55 所示。

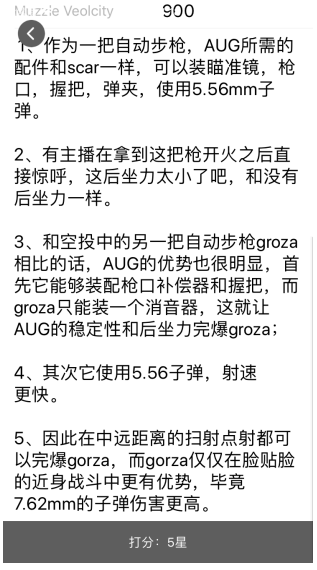


图 23-55

作业：用 Image View 来显示一星到五星，并加上合适的动画效果。

23.6　JSON 编/解码

至此，我们之前的 App 的 Model（数据）都是“硬编码”（写在代码的数据）。实际上 App 数据大多数来自网络，并且是动态变化的，是“软”的。这就不得不提到 JSON 了，实际上无论我们做什么编程开发，可能都要学习它。下面我们就来看看在 Swift 开发中 JSON 有什么特色。

先来看下面的一幅图，如图 23-56 所示。

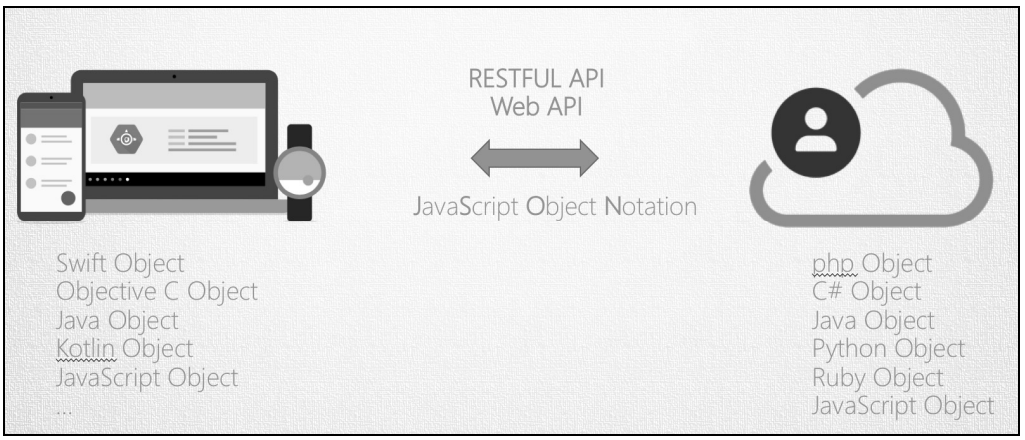


图 23-56

JSON 的全称是 JavaScript Object Notation，即 JavaScript 对象表示法。虽然它是 JavaScript 的范畴，但普遍被用于各种客户端程序（Windows/Mac、Web、移动设备）与服务器端进行通信的接口媒介。

让所有计算机都安装任何一种高级语言编程开发工具是不可能完成的任务。浏览器每台计算机都有，可能还有好几个，而任何一个浏览器都自带 JavaScript 开发环境，这就让它成了门槛为 0 的编程语言。几乎所有高级编程语言都支持面向对象特性，用 JavaScript 的对象做数据中转似乎也就“顺理成章”了。

读者可能经常看到 Web API、RESTFUL API 之类的名词，就是指这种数据是 JSON 格式的。

既然用 JSON 做中转，就要涉及编码和解码。服务端把数据编码成 JSON，客户端把 JSON 解码还原。

1. JSON 编码

本书不讨论服务端编程语言如何进行 JSON 编码的问题，但先来看看如何把 Swift 版本的武器库数据转换成 JSON，这就等于实现了服务端数据接口的模拟。

我们很幸运，从 Swift 4 开始，就有了专门的 JSON 编/解码类，用起来还是非常方便的。

任何类型只要遵从 Codable 协议，都会自动获得 JSON 编/解码功能，如表 23-14 所示。

表 23-14

类	作　用	适用类型
JSONEncoder	对象转 JSON	遵从 Codable 协议
JSONDecoder	JSON 还原	

给我们的武器 Model 加上 Coable 协议：

```
    struct Weapon: Codable {
    var name = ""
    var type = ""
    var origin = ""
    var image = ""
    var favorite = false
    var bullet = 0.0
    var mv = 0
    var text = ""
    var rating = ""


    init(name: String, type: String, origin: String, image: String, favorite:
Bool, bullet: Double, mv: Int, text: String) {
    self.name = name
    self.type = type
    self.origin = origin
    self.image = image
    self.favorite = favorite
    self.bullet = bullet
    self.mv = mv
    self.text = text
        }
    }
```

现在我们在主场景控制器 WeaponsTableViewController.swift 中加入一个方法，把 Weapons 数组转换为 JSON 并保存到文件中，完整代码如下。

```
    func saveToJson()  {
    let coder = JSONEncoder()

    do {
    let data = try coder.encode(weapons)
    let saveUrl = URL(fileURLWithPath: NSHomeDirectory()).appendingPathComponent
("weapons.json")

    try data.write(to: saveUrl)
```

```
    print("保存成功！路径是：",saveUrl)
          } catch  {
    print("编码错误：",error)
          }
    }
```

第一句代码是实例化一个 JSON 编码类，因为解析 JSON 可能因为各种原因而失败，所以加入 do/catch 代码块来处理可能出现的错误。尝试把 Weapons 数组编码成 JSON 格式的数据。保存在 App 根目录下，文件名为 weapons.json，然后写入代码。

这个方法放到 viewDidLoad 中调用，然后运行。

如果以上代码运行没有错误，我们会在控制台看到如下文字：

```
保存成功！路径是：file:///Users/yons/Library/Developer/CoreSimulator/Devices/4E17C1F1-95D3-4582-A68D-5B3291549F18/data/Containers/Data/Application/41C4A3BE-47E6-49D3-806E-9494453E7783/weapons.json
```

因为模拟器的文件系统也在 macOS 系统内，所以我们可以打开 Finder（新版本中叫访达）直接跳到这个文件夹内核实文件生成情况，如图 23-57 所示。注意，路径中的一长串十六进制的目录名是随机的，可能每次运行都会不同。



图 23-57

进入 Finder 的前往文件夹功能，删除 file://和文件名的中间一段，这是系统可以接受的路径格式，即：

```
/Users/yons/Library/Developer/CoreSimulator/Devices/4E17C1F1-95D3-4582-A68D-5B3291549F18/data/Containers/Data/Application/41C4A3BE-47E6-49D3-806E-9494453E7783/
```

我们会发现这个多级目录下有 weapons.json 文件，如图 23-58 所示。这说明我们的代码编写成功。

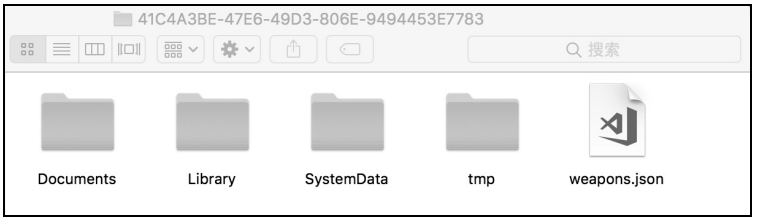


图 23-58

可以看到，这个目录下还有几个文件夹，这是 App 存储数据的地方。tmp 存放临时文件，Documents 一般用来存放常用的数据，如用户登录过的名称及密码、之前的历史记录、缓存等。iOS 不允许 App 存储和读取任何此目录以外的数据，这就是著名的“沙盒”机制，从而保证了隐私不会被恶意 App 窃取，当然“越狱”过的系统没有此限制。

打开 weapons.json，整体内容是一行到底的，如果安装了 Visual Studio Code 这样的高级编辑器，可以将内容格式化，显示为规整的 JSON 格式，如图 23-59 所示。

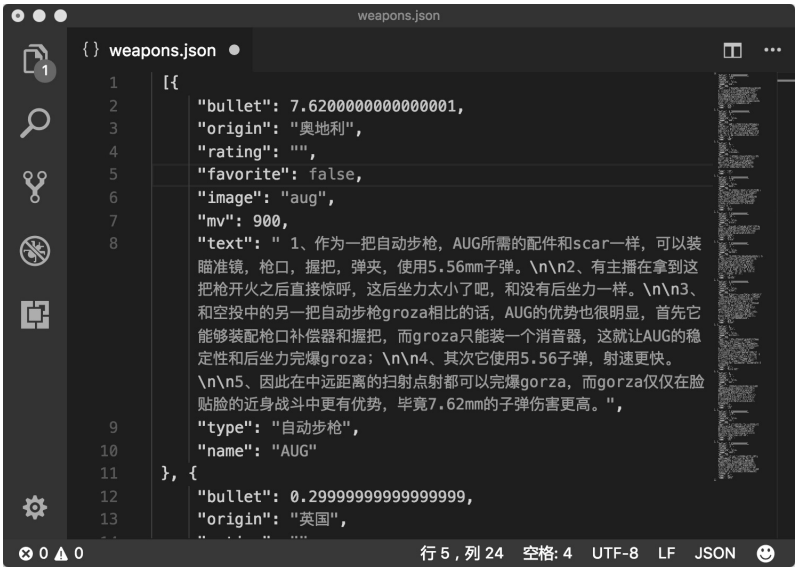


图 23-59

可以看到，每个对象被包含在一对花括号中，属性和值用逗号分隔，多个对象的花括号也是用逗号分隔的，如果是个数组，则用一对分号包含。

把这个文件添加到我们的 XCode 工程中，勾选 Copy items if needed 复选项，如图 23-60 所示。

这样 weapons.json 就成了 App 工程包自带的数据文件了，可以随时读取，如图 23-61 所示。

2. JSON 解码

现在终于可以把 weapons 数组留空了，saveToJson 方法的调用也可以“功成身退”了。

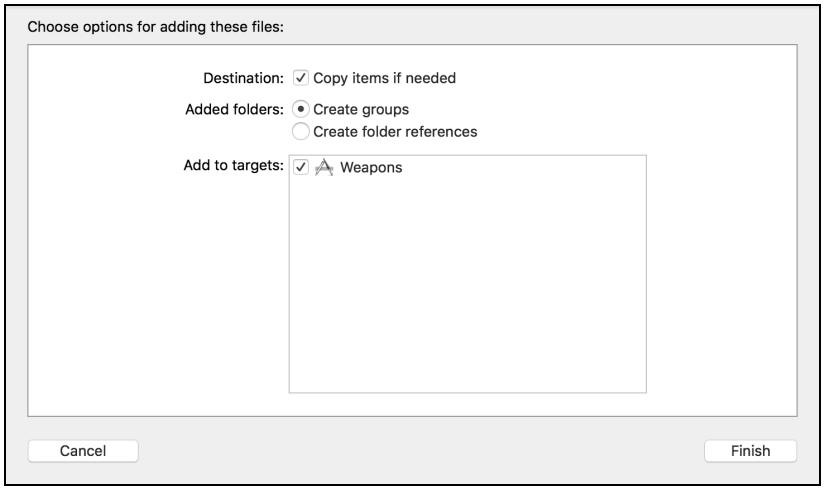


图 23-60

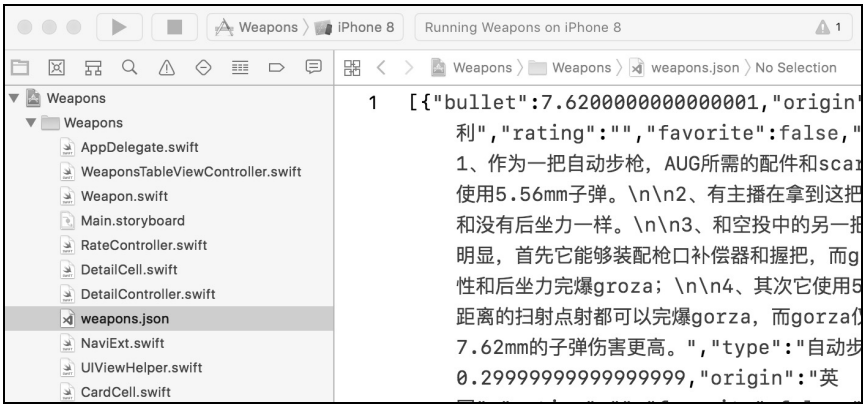


图 23-61

```
var weapons: [Weapon] = []
```

编写一个 loadJson 方法，用于读取 json 文件并解码到 weapons 中：

```
func loadJson()  {
let coder = JSONDecoder()

do {
let url = Bundle.main.url(forResource: "weapons", withExtension: ".json")!
let data = tryData(contentsOf: url)

weapons = try coder.decode([Weapon].self, from: data)
      } catch  {
print(error)
      }
   }
```

第一句代码是实例化 JSONDecoder 解码器。与编码时一样，使用 do/catch 代码块处理错误。

获取 weapons.json 文件的 URL，Bundle.main 是 App 工程包的根路径。然后尝试读取这个文件，转为数据，最后尝试用解码器解码为 weapons 数组的实例，用来更新控制器的数据源。

那么何时解析这个 JSON 呢？当然是在主场景启动时解析，因为需要让 Table View 来显示武器数据，而现在数据源 weapons 数组已经是空的了：

```
overridefunc viewDidLoad() {
super.viewDidLoad()

navigationController?.navigationBar.largeTitleTextAttributes = [
NSAttributedStringKey.foregroundColor : UIColor(named: "theme")
       ]

loadJson()
    }
```

运行后，我们会发现一切正常，武器数据照常显示。

至此，读者应对 JSON 在数据传输中的作用有了一个直观的了解。

作业：尝试给武器模型增加一些属性，再转换成 JSON 格式。

23.7 录入新数据场景

之前的数据都是我们手工制造的，没有一个让用户录入的交互入口。本节将创建一个可以让用户方便录入新绝对求生武器的场景。

录入场景的头部和尾部完成的效果如图 23-62 和图 23-63 所示。



图 23-62

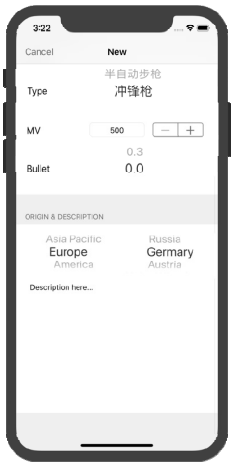


图 23-63

1. 新场景布局

由于武器的参数比较多，为了方便，我们可以使用 Table View Controller 把参数分组显示，以供用户更方便地录入。

新场景的配置如表 23-15 所示。

表 23-15

Table View Controller		
Table View	Content	Static Cells
	Separator	None
	Sections	3
	Style	Grouped

Sections 的配置如表 23-16 所示。

表 23-16

Sections	
Header	Rows
Photo/Video select...	1
Specification	4
Origin & Description	2

新场景的 Table View 结构如图 23-64 所示。

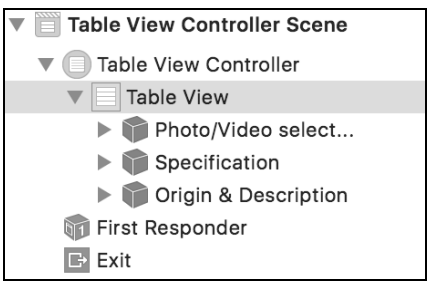


图 23-64

由于新场景上的 UI 元素可能有很多，默认的宽度在 Storyboard 上不能全部预览，可在 Controller 的尺寸属性中把模拟尺寸调整为 Freedom，高度调整为 1100，如图 23-65 所示。

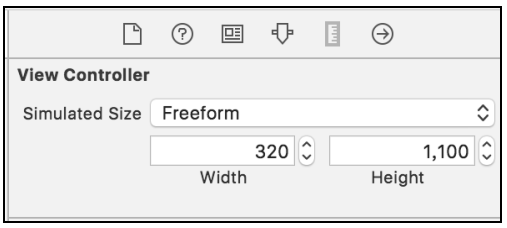


图 23-65

来到 Section 1 的布局，由 header 名可知，这个 Section 是用来让用户选择照片或视频的，选完之后必然有个预览区，所以放一个 Image View 上去，布局如表 23-17 所示，预览如图 23-66 所示。

表 23-17

Row Height			200
子视图	Image View	Image	gun-placeholder2
		Content Mode	Aspect Fill
		约束	水平垂直居中

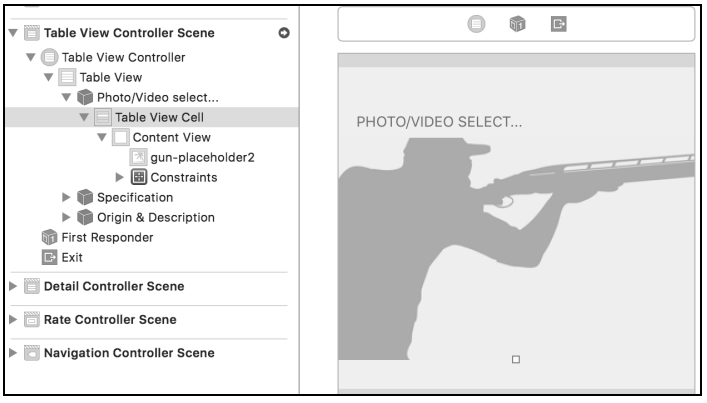


图 23-66

Section 2 用于输入武器的各种参数，各 Cell 布局如表 23-18 所示，预览如图 23-67 所示。

表 23-18

Cell 1 – 名称输入				
Stack View	子视图	Label	Title	Name
		Text Field	Placeholder	Weapon name here...
	属性	Spacing	15	
	约束	Y	垂直居中	
		X	左右边距	20
Cell 2 – 武器类型				
Row Height			100	
Stack View	子视图	Label	Title	Type
		Picker View	tag	10
			datasource delegate	table view controller
	属性	Spacing	15	
	约束	Y	垂直居中	
		X	左右边距	20

（续表）

Cell 3 – 射速				
Stack View	子视图	Label	Title	MV
		Text Field	Text	500
		Stepper	step Max Min	50 2500 500
	属性	Spacing	15	
		Distribution	Equally	
	约束	Y	垂直居中	
		X	左右边距	20
Cell 4 – 子弹类型				
Row Height			100	
Stack View	子视图	Label	Title	Bullet
		Picker View	tag	11
			datasource delegate	table view controller
	属性	Spacing	15	
	约束	Y	垂直居中	
		X	左右边距	20

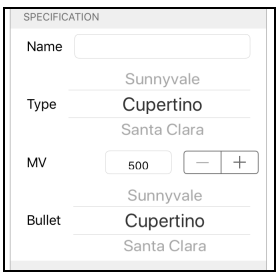


图 23-67

Section 3 作为产地选择和武器描述录入之用，两个 Cell 配置如表 23-19 所示，预览如图 23-68 所示。

表 23-19

Cell 1 – 产地选择			
Row height		100	
子视图	Picker View	tag	12
		datasource delegate	table view controller
	约束	左右边距	20
		上下	0

（续表）

Cell 2 – 描述多行文本框			
Row height		250	
子视图	Text View	Text	Description here...
	约束	左右边距	20
		上下	0

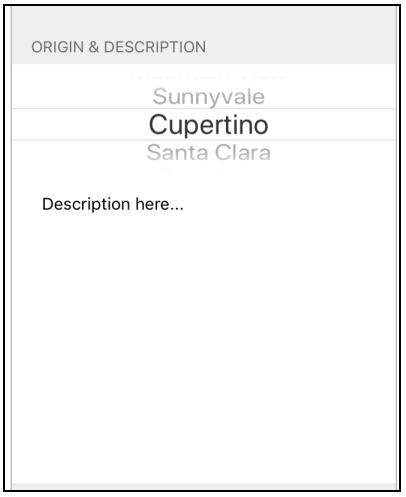


图 23-68

是否感到心力交瘁？这么多的内容。我们毕竟还没有开始编写代码。

为了交互方便，给新场景嵌入导航，在导航栏上添加 Bar Item，用系统提供的 Cancel 作为取消按钮，如图 23-69 所示。

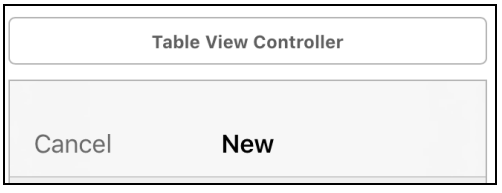


图 23-69

首页导航右侧加一个按钮（System Item 是 Add），并拖曳一个转场到新场景中，这样录入数据就有了入口，如图 23-70 所示。



图 23-70

转场参数如表 23-20 所示。

表 23-20

Segue	到“New”场景的 Navigation Controller
Kind	Present Modally
Identifier	addNew

转场属性设置画面如图 23-71 所示。

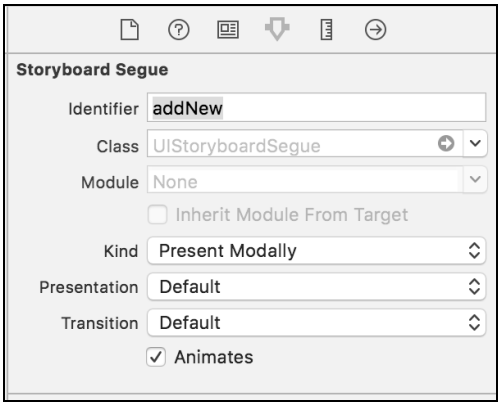


图 23-71

添加新数据既然有了入口，就要有出口，单击 Cancel 按钮后要能够返回到首页。之前我们讲过的反向转场，需要在要返回的场景控制器中加入相关的代码，即在 WeaponsTableViewController.swift 中加入以下方法：

```
@IBActionfunc backToHome(segue: UIStoryboardSegue)  {


}
```

拖曳 Cancel 按钮到 Exit 处，选择 backToHomeWithSegue 菜单项即可。

2. 载入数据

我们希望从 JSON 来读取几个 Picker View 的选项，一是方便用户，二是可以消除输入错误。

像解析 JSON 的步骤一样，把预备好的 origin.json 和 weaponSpecs.json 加入工程，并在 weapon.swift 中加入相应的解析模型：

```
struct WeaponSpecs: Codable {
var types = [String]()
var bullets = [Double]()
}

struct Origin: Codable {
var continent = ""
var countries = [String]()
}
```

给新场景一个控制器，继承自 UITableViewController，命名为 NewWeapon Controller，并与 Storyboard 连接，因为我们用的是静态单元格，所以把代码中默认自带的数据源方法删除。

加入两个属性用来保持 JSON 解析后的值：

```
    var origin = [Origin]()
    var specs : WeaponSpecs!
```

在控制器中加入一个 loadJson 方法，同时解码两个 JSON 文件，可以参照 JSON 的分析方法，非常简单：

```
       func loadJson() {
           let coder = JSONDecoder()

           do{
             let url1 = Bundle.main.url(forResource:"origin", withExtension:
".json")!
              let url2 = Bundle.main.url(forResource:"weaponSpecs", withExtension:
".json")!
              let data1 = try Data(contentsOf: url1)
              let data2 = try Data(contentsOf: url2)

              origin = try coder.decode([Origin].self, from: data1)
              specs = try coder.decode(WeaponSpecs.self, from: data2)
              print("载入数据源成功！")
           } catch {
              print(error)
           }
       }
```

记得在场景载入时加入 JSON 解码方法：

```
    overridefunc viewDidLoad() {
    super.viewDidLoad()

    loadJson()
        }
```

3. Picker View

此时虽然 JSON 解码成功，但因为 Picker View 还没有填充数据，运行还是会报错的。

为了让控制器代码分离，做一个 extension 来处理 Picker View 的填充，即 pickerViewFill.swift。与 Table View 相似，Picker View 也有数据源和代理协议要实现，代码如下。

```
    extensionNewWeaponController: UIPickerViewDataSource ,UIPickerViewDelegate
{
    func numberOfComponents(in pickerView: UIPickerView) ->Int {
    switch pickerView.tag {
```

```
    case10:
    return1
    case11:
    return1
    default:
    return2
           }
       }
    }
```

numberOfComponents 代理方法是指显示几列（可以上下拨动的文字选项列）。因为场景上有三个 Picker View，之前我们给每个都定义了 tag 值，这里做了区分。除选择国家有两列外，前两个 Picker View 只有一列。

在上面的方法之后，加入另一个代理方法：

```
    func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent
component: Int) ->Int {
    switch pickerView.tag {
    case10:
    returnself.specs.types.count
    case11:
    returnself.specs.bullets.count
    default:
    if component == 0 {
    returnself.origin.count
             } else {
    returnself.origin[pickerView.selectedRow(inComponent: 0)].countries.count
             }
           }
       }
```

numberOfRowsInComponent 代理方法是指每列的行数，分别是数据源中对应的行数。产地的第二列稍有不同，它的列数与第一列的选择有关，因为每个洲对应的国家数目都不同，其中有一个联动。

此时运行，会发现 Picker View 里面都是问号。

因为数据源的代理方法仅正确实现了数目，但内容并没有设定。如果没有给定内容，默认就显示问号。Picker View 并没有像 Table View 那样强制要求我们实现每行具体的内容。不过，这实现起来也不难。

```
    func pickerView(_ pickerView: UIPickerView, titleForRow row: Int,
forComponent component: Int) ->String? {
    switch pickerView.tag {
    case10:
    returnself.specs.types[row]
```

```
    case11:
    returnself.specs.bullets[row].description
    default:
    if component == 0 {
    returnself.origin[row].continent
              } else {
    returnself.origin[pickerView.selectedRow(inComponent: 0)].countries[row]
              }
          }
       }
```

titleForRow_forComponent 是实现每列中每行内容的方法。

还差最后一步，滑动选择了洲后，对应的国家列表页需要改变。

```
    func pickerView(_pickerView:UIPickerView,didSelectRow row:Int,inComponent
component:Int){
    switch pickerView.tag{
    case12:
    if component == 0{
                pickerView.reloadComponent(1)
              }
    default:
    break
          }
       }
```

运行后，我们就能看到本节开始的 Picker View 显示效果了。

4. Stepper

最后设置一个 Stepper，要实现它与同单元格内文本框的联动非常简单。

Stepper 可以像 Button 那样使用，拖曳一个 IBAction 即可，完整代码如下。

```
    @IBOutletweakvar textfieldMv: UITextField!

    @IBActionfunc tapStepper(_ sender: UIStepper) {

    textfieldMv.text = Int(exactly: sender.value)!.description
       }
```

因为 Stepper 的 value 是 Double 类型，要做到 Int 的转换，基本类型的 description 属性就是用来转字符串的，简便易用。

作业：

①实现 3 级联动。②导航条右侧增加一个“保存”按钮，并判断文本框未输入的情况。③输入完成时，隐藏输入法键盘。

23.8　真机部署

至此，读者一定很想在 iPhone 或 iPad 上真正测试一下 App，本节就来介绍如何把 App 部署到真机上测试，也是为了后面的拍照和录制视频功能做准备。

1. 开发证书

需要证书的原因很简单：安全。要部署 App 到真机，代码都需要签名。证书分为开发证书和发布证书，只要有 AppleID 就可以免费申请苹果开发者，从而可以利用 XCode 把 App 部署到真机上。当然读者最终还是想上架到 AppStore 供全世界用户下载，那就要付费升级成苹果认证开发者（年费 688 元人民币）。

在苹果开发者官网（developer.apple.com）申请开发者账号后，就可以在 XCode 中登录使用了。选中工程的 target，执行菜单命令 General→Signing→Team→Add an account，弹出开发者账号页面，如图 23-72 所示。



图 23-72

这里再复查一下工程的 BundleID，确保不会和已有的 BundleID 冲突，因为这是全球唯一的标识符，如表 23-21 所示。

表 23-21

Bundle Identifier	
作用	App 的唯一标识符，全球唯一
命名规范	国际域名倒序 com.xiaoboswift.Weapons

XCode 可以自主管理开发证书、BundleID，勾选 Automatically manage signing 复选项，如图 23-73 所示。

2. 描述文件

这里还有一个术语，即描述文件。这个文件记录了一个 App 在真机运行的配置过程，用于全体组员在设备上对 App 进行签名和运行。它包含开发证书、AppID（与 BundleID 是一个意思）和授权可以运行的设备 ID（即 UDID）列表。

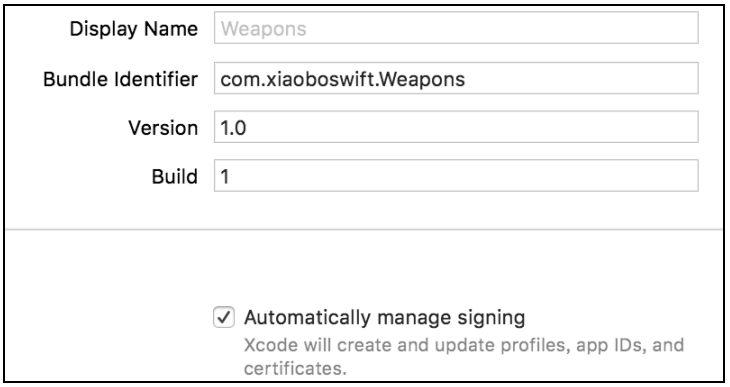


图 23-73

连接 iPhone 或 iPad 到 Mac，会在运行 Scheme 菜单中显示。如果没能显示，可能是设备的 iOS 版本比较低。

在自动签名选项的下方有一个 ProvisioningProfile，单击旁边的“i”符号按钮，即可看到描述文件包含的内容。

Xcode 自动管理的描述文件如下。

App ID：与 Bundle Id 匹配，*是通配符。

Certificates：包含开发证书。

Team：所在团队。

Capabilities：增强功能。

Entitlements：增强功能相关的授权证书。

Devices：此 Team 已授权的测试设备数。

真机运行准备就绪。运行 Scheme 选择自己的 iPhone/iPad，然后运行。设备必须处于解锁状态。但这时依然不能运行，如图 23-74 所示。



图 23-74

默认的开发者不受设备信任，必须手工添加信任。执行设备的菜单命令“设置”→“通用”→“设备管理”，找到相应的开发证书，单击“信任<开发者 email>按钮”，然后再次运行 App 即可。

3. 无线调试

XCode 9 开始支持无线调试，即只要连接一次真机，开启无线调试后，以后这台机器就可以不用连接 USB 进行调试了，不过前提是手机和 Mac 在同一个 WiFi 网络下。设备菜单如图 23-75 所示，开启无线调试如图 23-76 所示。

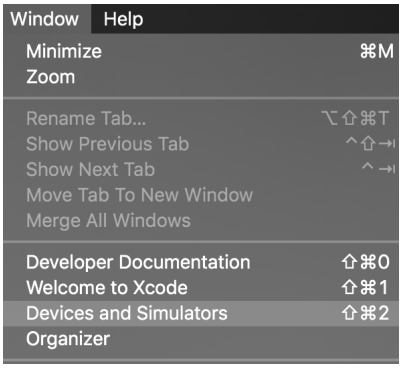


图 23-75

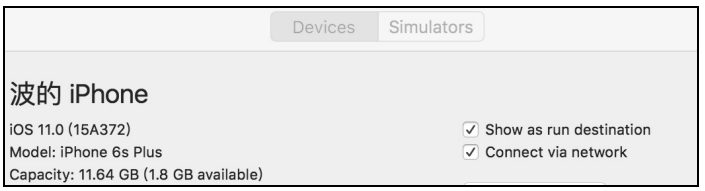


图 23-76

23.9　相册、拍照和录制视频

本节来实现相册、拍照和录制视频的交互和功能，这看起来很复杂但很有挑战性，以此来作为中级篇的完结。

1. 单元格点击

还记得首页的卡片中爱心按钮的交互吗？现在是单击单元格中的图片占位符来选择照片或者拍照，也可以用单元格整体交互。Table View 的 UITableViewDelegate 协议就具备这个特性，为了让整体代码模块化，新建一个 MediaSelect.swift 来扩展 NewWeaponController，代码如下。

```
    overridefunc tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
```

```
    print(indexPath)
           tableView.deselectRow(at: indexPath, animated: true)
    }
```

didSelectRowAtIndexPath 方法就是选择了单元格，参数 indexPath 代表单元格所在的位置。deselectRow 方法把选中的状态取消，所以单击时会有一个选中的闪烁特效。

这里其他单元格不需要单击交互的代码，所以加一个判断代码：

```
    if indexPath.section == 0 {
    print(indexPath)
    }
```

2. UIAlertController

通常让用户选择某几项操作时会用到菜单，在 iOS 中 UIAlertController 就是这种常用的菜单之一，可以从屏幕中央弹出（Alert），也可以从底部弹出（Actionsheet）。

Alert 样式如图 23-77 所示，Actionsheet 样式如图 23-78 所示。

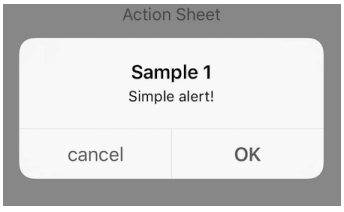


图 23-77

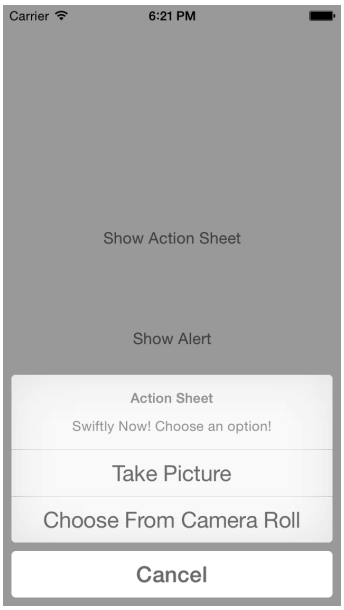


图 23-78

这两种操作的主要区别就是 ActionSheet 显示多个选项时比较好用，因为有足够的空间

可以显示，这里我们用一个 Actionsheet 样式的 UIAlertController 来显示菜单：

```
    letactionSheet    =    UIAlertController(title:    nil,    message:    nil,
preferredStyle: .actionSheet)

    letphotoAction = UIAlertAction(title: "Photo", style: .default, handler:
{ (_) in

    })

    lettakePhotoAction = UIAlertAction(title: "Take a photo", style: .default,
handler: { (_) in

    })

    letvideoAction = UIAlertAction(title: "Video", style: .default, handler:
{ (_) in

    })

    letrecordAction = UIAlertAction(title: "Record a video", style: .default,
handler: { (_) in

    })

    let cancelAction = UIAlertAction(title: "Cancel", style: .cancel)

    actionSheet.addAction(photoAction)
    actionSheet.addAction(takePhotoAction)
    actionSheet.addAction(videoAction)
    actionSheet.addAction(recordAction)
    actionSheet.addAction(cancelAction)

    present(actionSheet, animated: true)
```

第一句代码是实例化 actionSheet 样式的 UIAlertController，标题和消息这两个参数是可选的，所以是 nil。

下面的几个 UIAlertAction 是菜单项的定义，title 是菜单项的标题，取消按钮的样式是 cancel，这样可以自动位于屏幕下方并与其他菜单项隔离开。handler 参数是指单击后要执行的操作，这里先留空。

然后是把这些菜单项添加到 actionSheet 中，最后展示出来，效果如图 23-79 所示。

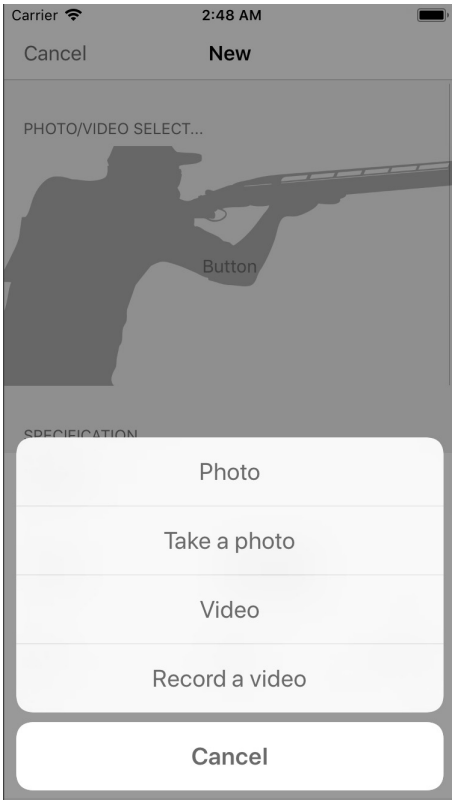


图 23-79

3. UIImagePickerController

说到拍照、录像、查看照片，读者自然会想 iOS 系统自带的照片和相机 App，而且已经非常熟悉 UI 及其操作了。没错，我们不需要自己发明，iOS 提供 UIImagePickerController 可以让我们轻松地使用这些功能。

首先要注意的是，由于相册、摄像头和话筒属于用户隐私，iOS 权限控制非常严格，必须明确地获取用户许可后才可访问。

在工程的 Info.plist 中增加 3 项，权限设置如图 23-80 所示。

Privacy - Camera Usage Description	String	需要使用摄像头拍照
Privacy - Photo Library Usage Description	String	需要打开相册
Privacy - Microphone Usage Description	String	录制视频时需要打开麦克风

图 23-80

接着依次实现各菜单项的功能，打开相册，也就是 photoAction，代码如下：

```
    let photoAction = UIAlertAction(title: "Photo", style: .default, handler:
{ (_) in
    guardUIImagePickerController.isSourceTypeAvailable(.photoLibrary) else {
    return
```

```
                }

    let picker = UIImagePickerController()
                picker.sourceType = .photoLibrary
                picker.delegate = self
    self.present(picker, animated: true)

            })
```

guard 语句部分的代码保证已经获取了相册访问权限，否则返回不继续执行。如果有了权限，实例化 UIImagePickerController，设置来源类型是相册，picker 的相关操作代理（选择了图片）是控制器，并展示出来。

效果如下图 23-81 所示。

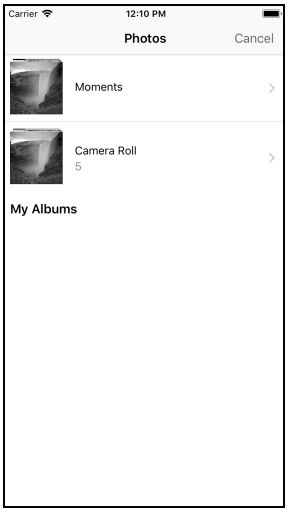


图 23-81

默认的模拟器中只有几张自带的图片，我们可以从 Finder 中拖曳图片到模拟器中，即可直接在相册中选择使用，如附件中的 akm.jpeg 和 p18c.jpg。

```
    let takePhotoAction = UIAlertAction(title: "Take a photo", style: .default,
handler: { (_) in
    guardUIImagePickerController.isSourceTypeAvailable(.camera) else {
    return
                }

    let picker = UIImagePickerController()
                picker.sourceType = .camera
                picker.delegate = self
    self.present(picker, animated: true)
            })
```

访问相机授权如图 23-82 所示。

拍照效果如图 23-83 所示。



图 23-82



图 23-83

选取视频文件也属于访问相册，只不过加了一个媒体类型筛选，这个类型在 MobileCoreServices 框架中定义，所以要在控制器代码顶端加入：

```
    import MobileCoreServices
```

完整代码如下。

```
    let videoAction = UIAlertAction(title: "Video", style: .default, handler:
{ (_) in
    guardUIImagePickerController.isSourceTypeAvailable(.photoLibrary) else {
    return
                }

    let picker = UIImagePickerController()
                picker.sourceType = .photoLibrary
                picker.mediaTypes = [kUTTypeMovieasString]
                picker.delegate = self
    self.present(picker, animated: true)
            })
```

在真机上运行就可以只选择视频文件了，不过在模拟器上默认没有视频，同样，可以直接在 Finder 中拖曳一个视频进入模拟器，如附件的 big_buck_bunny.mp4、pubg_how_to_aim.mp4、pubg_china_number_one.mp4。

选取视频效果如图 23-84 所示。



图 23-84

最后录制一个短视频，录制属于访问摄像头，与拍照代码相似，不同的是媒体类型，还有录制的品质和时限。完整代码如下。

```
    let recordAction = UIAlertAction(title: "Record a video", style: .default,
handler: { (_) in
    guardUIImagePickerController.isSourceTypeAvailable(.camera) else {
    return
                }

    let picker = UIImagePickerController()
                picker.sourceType = .camera

                picker.mediaTypes = [kUTTypeMovieasString]
                picker.videoQuality = .typeHigh
                picker.videoMaximumDuration = 10

                picker.delegate = self
    self.present(picker, animated: true)
      })
```

因为录制视频会用到话筒，所以会有授权提示，如图 23-85 所示。

录制效果如图 23-86 所示。



图 23-85



图 23-86

4. 取回照片和视频

下面该把照片和视频取回了，读者应该猜到 picker 的 delegate 的作用了。在此之前，为了显示选择的图片，以及增加一个播放视频的按钮（居中显示，尺寸为 100×100 点，默认 hidden 不显示）和导入附件中的播放按钮图片 play.png（3 种尺寸），如图 23-87 所示。

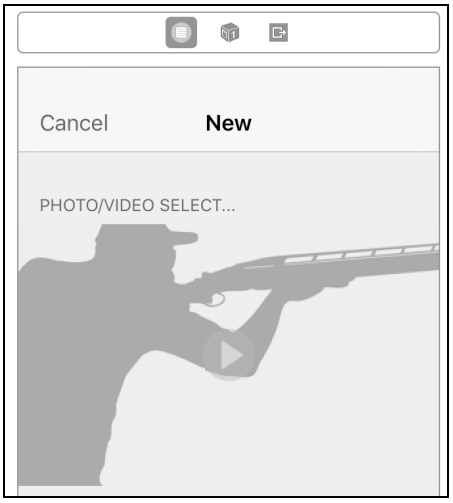


图 23-87

拖入图片和按钮到代码中以便引用：

```
    @IBOutletweakvar bgImage: UIImageView!
    @IBOutletweakvar playBtn: UIButton!
    var videoUrl: URL!
```

用一个属性来保持视频的 URL。

同样，为了使代码简洁，在 MediaSelect.swift 中扩展 NewWeaponController，因为 UIImagePickerController 自带一个导航控制器，所以让其同时遵守相册控制器的代理，以及导航控制器的代理。实现选择完图片或视频的代理方法，完整代码如下。

```
    extensionNewWeaponController: UIImagePickerControllerDelegate, UINavigation
ControllerDelegate {
    func imagePickerController(_ picker: UIImagePickerController, didFinish
PickingMediaWithInfo info: [String : Any]) {
    let mediaType = info[UIImagePickerControllerMediaType] as! String

    if mediaType == (kUTTypeMovieasString) {
    videoUrl = info[UIImagePickerControllerMediaURL] as! URL

    let asset = AVAsset(url: videoUrl)
```

```
    let gen = AVAssetImageGenerator(asset: asset)
    let time = CMTime(seconds: 0, preferredTimescale: 2)
    let image = try! gen.copyCGImage(at: time, actualTime: nil)
    bgImage.image = UIImage(cgImage: image)

    playBtn.isHidden = false
           } else {
    bgImage.image = info[UIImagePickerControllerOriginalImage] as? UIImage
           }

           picker.dismiss(animated: true)
       }
    }
```

代码解释：

（1）didFinishPickingMediaWithInfo 这个代理方法在已经选取相册中的媒体资源后被调用。

（2）从参数 info 这个字典中查询 key 为“媒体类型”的数据，转换为字符串。

（3）分别处理视频和图片。如果是图片，则查询 info 中的原始图片，直接给背景图片：如果是视频，则取得视频文件的 URL 并保存，并获得缩略图（需要导入 AVFoundation），让播放按钮可见。

（4）最后关闭 UIImagePickerController。

用真机和模拟器都测试一下。

如果选择的图片显示不正常，是因为图片缺少大小的约束，让其与 superView 等宽、等高即可。

不要忘记还有最后一步，选择或录制了视频，希望能播放，增加播放按钮的互动代码如下。

```
    @IBActionfunc playVideo(_ sender: Any) {
    let playerVC = AVPlayerViewController()
           playerVC.player = AVPlayer(url: videoUrl)

    self.present(playerVC, animated: true) {
              playerVC.player?.play()
           }
       }
```

AVPlayerViewController 是系统自带的标准视频播放控制器，它既可以播放本地视频，也可以播放网络视频，只要指定它的播放器组件的 URL 即可。播放视频效果如图 23-88 所示。



图 23-88

总结：本节我们构建了一个比较复杂的 UI 和各种 iOS 组件的应用，最后实现了全功能的照片、拍照、视频和录制功能，难度不小，这算是对之前所学知识的总结。

作业：尝试把新增的数据传回首页列表。

第 3 篇



游戏实战篇

第 24 章
卡牌斗兽棋

项目名称：卡牌斗兽棋
项目类型：小游戏
语言版本：Swift 4
编译器版本：Xcode 9.2

24.1 项目介绍与分析

1. 项目背景

项目灵感来自笔者儿时与小伙伴们玩的一款小游戏。找一块硬纸板剪成 16 个大小相同的正方形卡牌，在卡牌一面用两种颜色的笔分别写上象、狮、虎、豹、狼、狗、猫、鼠，然后在地上画上 4×4 的方格棋盘，就可以开始游戏了。

然而，现在各种高科技电子产品早已进入小朋友们的生活中了，所以笔者想将这个小游戏用 Swift 实现，同时作为项目实战案例进行分享。

当然，现在也不缺各种小游戏。写本案例的目的主要是希望能够给 Swift 的学习者们，从项目的角度来学习 Swift 4 的一些知识，同时也希望给 Swift 初级开发者们带来一些做项目时思考和分析问题的方式，还希望能够给开发者们带来一些如何更好地写出项目代码的方法。基于这样的目的，所以在案例设置过程中会穿插一些知识点和业务逻辑的分析及思考。但对于已经有很高水平的开发者来说，本案例可能就是班门弄斧了。总之，只要本案例能够给读者带来一定的收获，笔者也就倍感欣慰了。

2. 实现玩法介绍

玩法分为双人单机对战、人机对战、联网双人对战三种不同的模式。本案例将逐步实现不同的游戏模式。

3. 游戏规则介绍

游戏分为两种颜色的 8 种动物（象、狮、虎、豹、狼、狗、猫、鼠），共 16 张卡牌，开局时随机分布在 4×4 的正方形方格棋盘内，背面全部朝上。开局后，一名玩家随机翻开第一张卡牌，则翻开的卡牌颜色属于该名玩家。另一名玩家则拥有另一种颜色的卡牌。

走棋规则：两名玩家轮流走棋，可以选择翻棋，也可以选择走棋，走棋时可以选择上、下、左、右中的一个方向，每次只能移动一格。

吃子规则：按照（象、狮、虎、豹、狼、狗、猫、鼠）从大到小的顺序，可以吃不大于本身的棋子，但最小的鼠可以吃象，象不可以吃鼠。

4．项目功能分析

项目功能分析，在本案例中其实就是对整个游戏的对战过程涉及的业务逻辑进行简单梳理。整个游戏过程其实可以简单地用流程图表示，如图 24-1 所示。

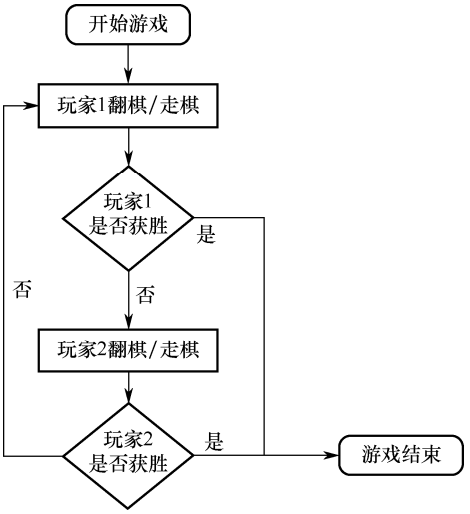


图 24-1

对项目中业务功能的需求也简单地做个思维导图，希望能够让读者更清晰地了解项目中需要实现的功能，如图 24-2 所示。

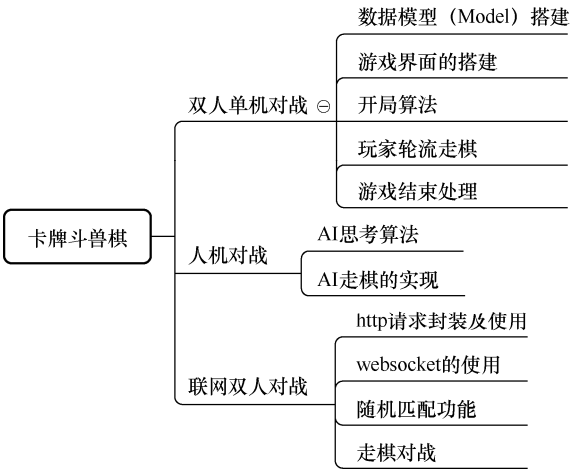


图 24-2

5．项目涉及的知识点

本案例主要从 Swift 4 的基本语法着手。不仅涵盖了 var、let、struct、class、enum、protocol、常见运算符、自定义运算符、闭包方法，以及 UIKit 中常见的一些控件等在项目中的基本使用，同时也包括了项目框架的搭建和项目开发过程中对问题的思考及分析。希望通过这样一个实战项目让读者能够更加扎实地掌握 Swift，并能更好地将 Swift 运用到以后的项目中去。

24.2　项目框架搭建

1．准备工作

打开 Xcode 9.0+的编辑器，新建项目。

（1）选择 Single View App，如图 24-3 所示。

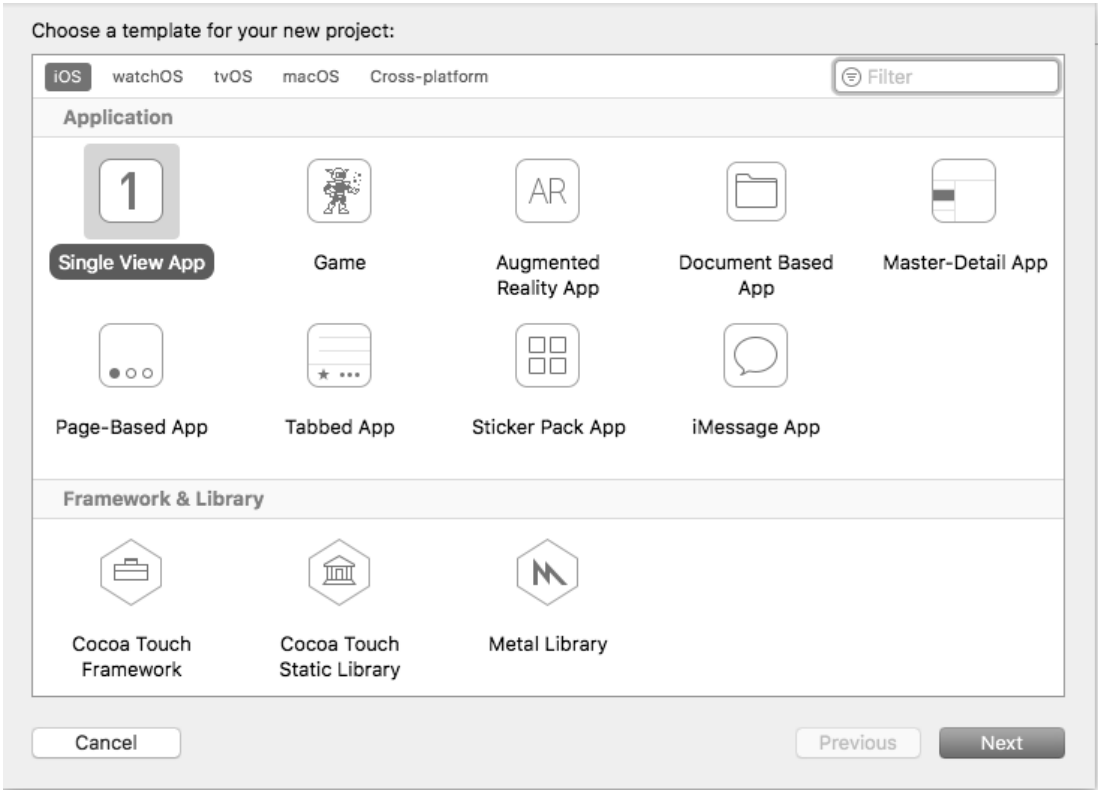


图 24-3

（2）填写项目名称：AnimalCardChess，如图 24-4 所示。

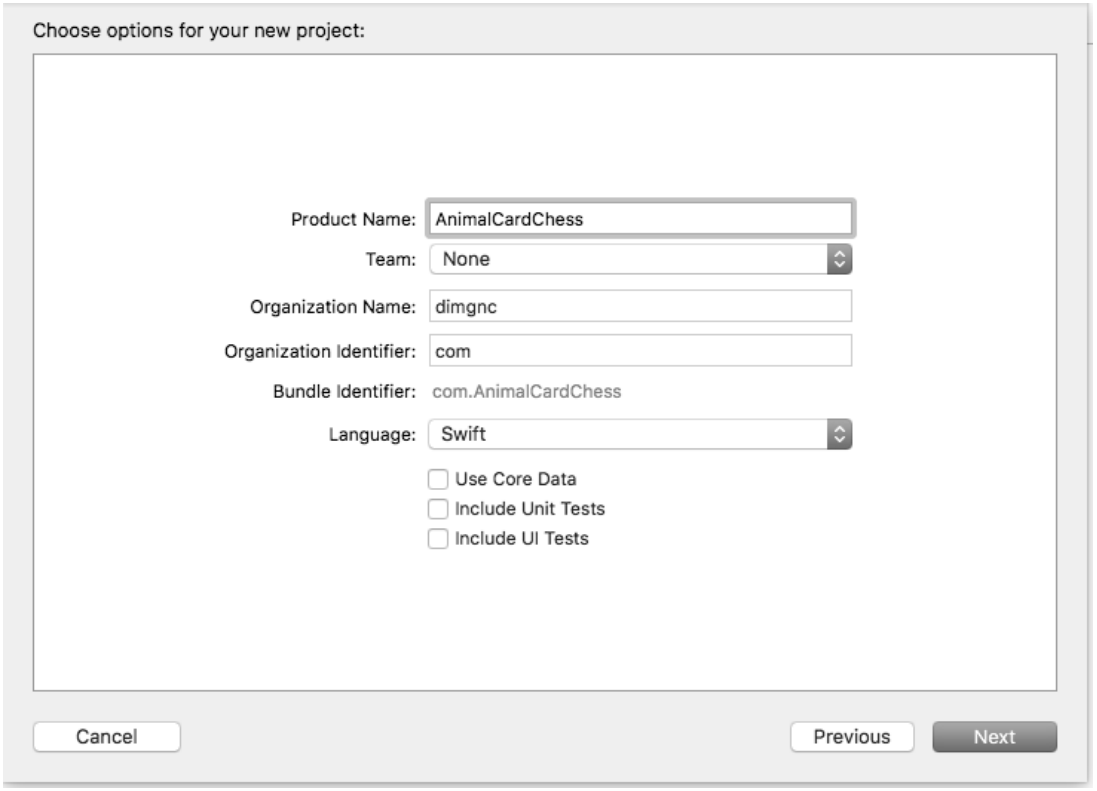


图 24-4

2．项目目录结构的介绍

一个项目在开始时，就应有一个清晰的目录结构。清晰的目录结构可让人明白相应的业务和职能，对项目的开发和后期维护都是非常重要的。

通常情况下，项目中有两种方式进行目录结构创建，这里以 MVC 模式为例，对两种目录结构简单地进行比较。

第一种：先按照业务分类，再按照项目模块分类，如图 24-5 所示。

第二种：先按照项目模块分类，再按照业务分类，如图 24-6 所示。

两种目录结构也是各有优缺点。第一种能够比较快地定位对应业务，但模块相关的类就比较分散，需要来回切换，且不利于多人合作开发。这种目录结构比较适合需求单一的小型项目开发。

第二种就是我们这个案例使用的目录结构，这种目录结构也是笔者比较喜欢使用的，它能够对模块进行集中化管理，且模块内可以多层嵌套模块。缺点就是有些共用的类不好归类。但是，某些共用的类可以单独提出来作为一个共用模块（Common）。这种结构适合较大型的项目开发，且利于团队分工开发，以及后期的版本迭代。

当然，实际项目中也并非绝对只有这两种固定的结构，很多时候可以对这两种结构进行一定的结合。这就需要笔者在实际开发中进行一定的探索和总结了。总之，没有最好的结构，只有最适合的结构。

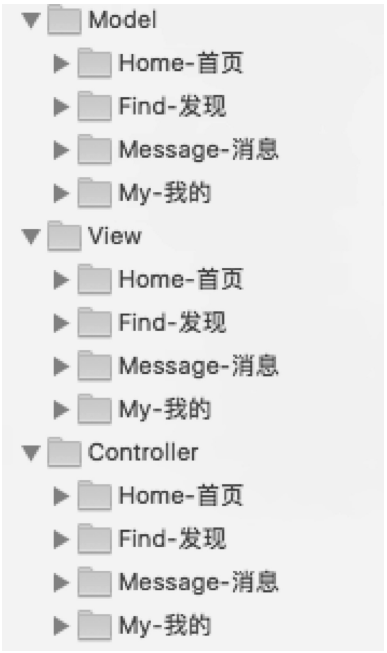


图 24-5



图 24-6

这两种项目结构的划分都涉及了 MVC 模式。MVC 也是软件项目开发过程中最常见的一种模式，关于这个模式的介绍也是随处可见的，这里就不赘述了。读者想要掌握其真谛，就必须经历很多项目实战。

3．项目目录结构的创建

我们先看结构图，这是项目的主目录，后面的开发将全部基于这个结构进行项目文件的创建，如图 24-7 所示。

各目录的作用如下。

Vender：这里存放的是一些第三方库，或者对第三方库的进一步封装。

Common：项目公共模块。下面又细分一些子目录。

Customer：这里存放项目中的一些全局用到的自定义视图，如弹出对话框。

Util：这里存放的是自己封装的一些工具类，如自己封装的网络请求库。

Extension：这里存放的是对系统提供的一些 Class 或 Struct 的扩展，如 Extension+UILabel。

General：这里存放的是项目中的一些常量，或者全局用到的一些样式，如字体大小、颜色等。

Base：这里存放的是项目中的基类，如 BaseViewController、BaseTabelViewCell。

Section：这里是存放项目功能模块的地方，其子目录也是按照前面提到的第二种目录结构进行划分的。

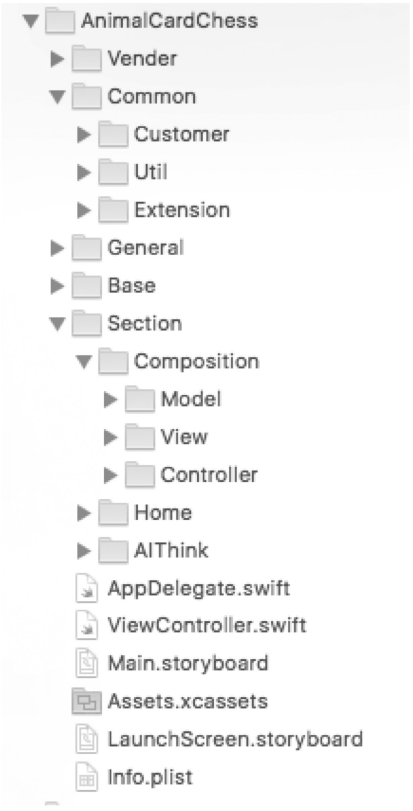


图 24-7

4．项目中基类控制器文件的创建

首先创建基类。在 Base 下创建 Controller 文件夹，在其下创建一个继承于 UIViewController 的 BaseViewController 类，如图 24-8 所示。之后项目中的控制器都继承于这个基类。

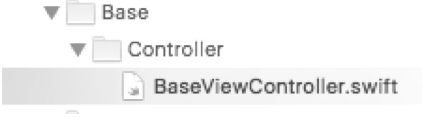


图 24-8

为何要这样做呢？这样方便对项目中的共性进行维护。例如，导航栏颜色、导航栏字体的样式、页面统一的背景色。我们可以在 viewDidLoad 里加入界面背景色的代码。

```
import UIKit
class BaseViewController: UIViewController {
  override func viewDidLoad() {
    super.viewDidLoad()
```

```
        self.view.backgroundColor = UIColor.lightGray
    }
    override func didReceiveMemoryWarning() {
    super.didReceiveMemoryWarning()
    }
}
```

关于继承和基类，有些需要注意的事项，也就是继承的深度和超级基类的问题。

继承的层次不能太深。有些时候，可能会发现需要写一个新的视图，与已经存在的视图里的控件差不多，只是布局有些偏差，然后就会去继承原有的类，重写一些方法。这样看上去好像省事了，但如果后期原有的那个视图需求有了大的改变，这时就会发现改起来涉及内容太多，于是就重写一个类，但原来的继承结构又放在那里成了多余的东西。

很多时候可能为了省事，就直接在一个比较高的基类里写入了一个少部分子类才会用到功能，这样随着项目功能的增加，就会出现一个庞大的基类，好像什么事都能处理了，所以也称之为“上帝类”。但这样一个基类，看似功能很强，实则很难维护，对于大型项目软件后期开发及维护是灾难性的。

对于继承这个问题，项目中肯定是避免不了的，对于继承层次的把握，最好不要将少数子类的功能实现在基类，而是使用封装的手段来实现。同时，尽量让继承的宽度代替继承的深度，一般的应用型 App 项目，尽量将继承层次控制在三层之内。

5．CocoaPods 的简单使用及第三方框架的导入

使用现成的第三方框架也是很多项目中不可少的一部分，使用好的第三方库也可以让我们的项目开发效率大大提升。这里以使用 SnapKit 为例，简单地介绍 CocoaPods 的使用，如何在项目中导入第三方框架。

什么是 CocoaPods 呢？CocoaPods 是 OS X 和 iOS 下的一个第三类库管理工具，通过 CocoaPods 工具，我们可以为项目添加“Pods”的依赖库（这些类库必须是 CocoaPods 本身所支持的），并且可以轻松管理其版本。

关于 CocoaPods 的安装教程，这里就不再详细介绍了。读者可以到网上搜索教程，这里推荐几个相关网址。

官网：https://cocoapods.org

相关教程：http://www.jianshu.com/p/f43b5964f582

SnapKit 将本案例中进行界面布局用到的第三方库（本案例布局并不没有使用 Storyboard，虽然它也是个很好的选择），在其官网上有着这样醒目的一句话：“SnapKit is a DSL to make Auto Layout easy on both iOS and OS X”。SnapKit 是让 iOS 和 OS X 自动布局更加简单的一个封装，并且 Snapkit 也早已经支持 Swift 4 了，可以在其 github 上查看对应的导入指令： pod 'SnapKit', '~> 4.0.0’。首先在项目目录下新建一个文本文件，命名为 Podfile，如图 24-9 所示。



图 24-9

用文本工具打开文件，在文件内写入相关导入代码：

```
platform:ios,'9.0'
use_frameworks!

target'AnimalCardChess'do
        pod'SnapKit','~>4.0.0'
end
```

打开终端，进入项目目录（cd 项目路径）。接下来输入 pod install 指令，等待导入完成。项目目录下就会多出几个文件，如图 24-10 所示。

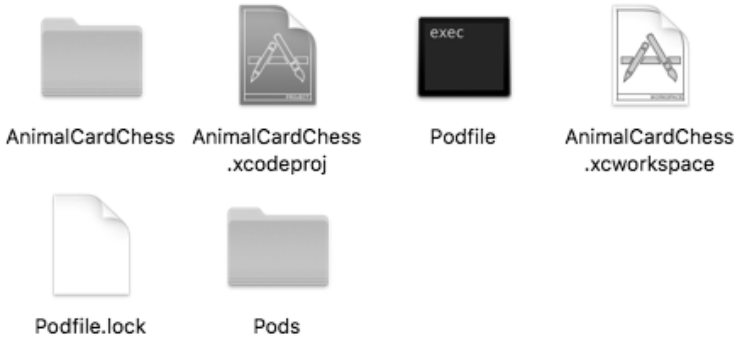


图 24-10

注意，有个白色背景和 Xcode 的 icon 的图标相似的文件，该文件就是 CocoaPods 导入时自动生成的，之后项目就直接通过这个文件打开。

24.3 首页和棋盘控制器的搭建

1. App 入口及首页控制器的搭建

在前面的准备工作完成后，就要开始搭建和游戏相关的界面了。

前面创建的那些空的文件夹目录，我们将从现在开始慢慢地让其丰满起来。

在 Section 的 Home 目录下创建首页 HomeViewController，这里就继承与我们前面创建的基类控制器 BaseViewController。HomeViewController 是应用打开时进入的第一个控制器，那么我们如何让应用一打开就进入这个控制器界面呢？只要在 AppDelegate.swift 文件中含有 didFinishLaunchingWithOptions 的函数里加入几行代码：

```
        func     application(_application:UIApplication,didFinishLaunchingWith
Options launchOptions:[UIApplicationLaunchOptionsKey:Any]?)->Bool{

      self.window=UIWindow(frame:UIScreen.main.bounds)
      self.window!.makeKeyAndVisible()
      self.window!.rootViewController = HomeViewController()
      return true
    }
```

接下来就是 HomeViewController 的具体功能的实现了，首先搭建首页界面，正式项目中这里肯定是会有设计图的。首页的主要功能也就是提供不同方式游戏的入口。三个入口分别为：双人对战、人机对战、联网对战，图 24-11 所示。



图 24-11

这个界面非常简单，应选用什么样的 UIKit 里的控件来实现呢？虽然用 UIButton 也是可以实现的，但这里我们并不推荐使用三个 UIButton，而推荐使用 UITbaleView 来实现这个界面。接下来通过这样一个简单的界面，让读者了解如何更好地在项目中使用 UITbaleView。当然这里基于读者已经掌握了 UITableView 的基础使用。

首先在 HomeViewController 类里定义一个 UITableView 控件。

```
    lazy var tv:UITableView={
let tv=UITableView()
tv.delegate=self
tv.dataSource=self
tv.backgroundColor=UIColor.clear
tv.separatorStyle=.none
tv.showsVerticalScrollIndicator = false
return tv
    }()
```

接下来让 HomeViewController 遵循 UITableViewDelegate、UITableViewDataSource 协议，并实现相关代理方法。

```
extension HomeViewController:UITableViewDelegate,UITableViewDataSource{
  func tableView(_tableView:UITableView,numberOfRowsInSection section:Int)->
Int{

  }

  func tableView (_tableView: UITableView, cellForRowAt indexPath: Index
Path)-> UITableViewCell{

  }

  func tableView(_tableView:UITableView,heightForRowAt indexPath:Index
Path)-> CGFloat{

  }

  func tableView(_tableView:UITableView,didSelectRowAt indexPath:Index
Path){

 }
}
```

下面将 UITableView 加入控制器的 UIView 中，并使用 SnaptKit 进行布局，我们可以编写两个方法。

```
        func createView(){
    self.view.addSubview(self.tv)
}

  func createSnp(){
    self.tv.snp.makeConstraints { (make) in
```

```
        make.center.equalToSuperview()
      make.width.equalToSuperview().multipliedBy(0.5)
        make.height.equalTo(150)
      }
    }
```

关于 SnaptKit 的布局具体使用，只要使用 view.snp.makeConstraints 方法的闭包里完成相关约束代码即可，如：

make.center.equalToSuperview()，添加当前与父视图居中的约束；make.width.equalToSuperview().multipliedBy(0.5)，添加当前控件宽度与等于父视图宽度一半的约束；make.height.equalTo(150)，添加当前控件高度等于 150 的约束。

完成 UITableView 的创建后，就要给 UITableView 添加数据源了，所谓数据源就是 UITableView 展示的数据都来自一个数组，那么这里的数据源就是按钮上的文字标题，可以用一个 String 数组，然后在 didSelectRow 代理方法里通过 indexPath.row 的值来判定单击了哪一个按钮。但这样做的话如果后期要把按钮的位置改变顺序，那么就需要修改数据源的顺序和 didSelectRow 方法里 indexPath.row 值判断的两个地方。在项目中，我们可以定义一个简单的 model，让代码后期维护变得更加简单。

例如，以下这样的代码：

```
struct HomeItem{
        var name:String
        var key:String
}
```

将这样的 model 作为数据源，就能够在 didSelectRow 里通过 key 的值来判断当前单击的是哪个元素，去除了单击方法与数据源顺序之间的耦合关系。另外，这样做还有其他好处，如设计图中要给每个按钮添加不同的图标，那么我们就可以直接在 model 里加个图标属性字段即可。

有了数据源，接下来就是具体的 UITableViewCell 的实现了。这里同样从项目的角度来看，实现一个自定义的 UITableViewCell，并对单击事件做一个简单的放大动画效果。首先创建一个继承于 UITableViewCell 的 HomeTableViewCell。此时 Home 下的文件如图 24-12 所示。

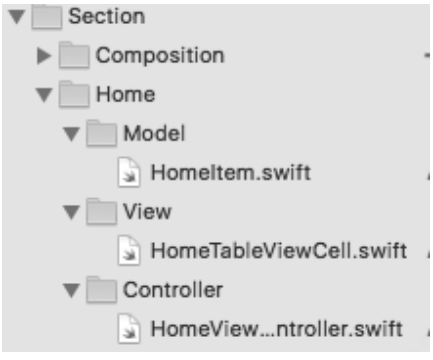


图 24-12

HomeTableViewCell 文件创建时选择继承于 UITableViewCell，则会自动生成如下代码。

```
import UIKit

class HomeTableViewCell:UITableViewCell{

  override func awakeFromNib(){
    super.awakeFromNib()
    //Initialization code
  }

  override func setSelected(_selected:Bool,animated:Bool){
    super.setSelected(selected,animated:animated)
    //Configure the view for the selected state
  }

}
```

分析这个 Cell 内的布局，按钮之间的间隔，同时需要实现一个当按钮被按下后放大的效果。我们可以在 Cell 里添加一个 UIView，以及一个 UILabel 来实现这样的布局：

```
var backView:UIView!
var nameLabel:UILabel!
```

重写 UITableViewCell 的初始化方法，将控件实例化且加上相应的样式，并使用 SnapKit 进行相应的布局约束：

```
override init(style:UITableViewCellStyle,reuseIdentifier:String?){
    super.init(style:style,reuseIdentifier:reuseIdentifier)
    self.backgroundColor=UIColor.clear
    self.selectionStyle=UITableViewCellSelectionStyle.none

    self.backView=UIView()
    self.backView.backgroundColor=UIColor.white
    self.contentView.addSubview(self.backView)

    self.nameLabel=UILabel()
    self.nameLabel.textAlignment=.center
    self.nameLabel.textColor=UIColor.init(red:0.3,green:0.3,blue:0.3,
alpha:1)
    self.nameLabel.font=UIFont.systemFont(ofSize:15)
    self.backView.addSubview(self.nameLabel)
    self.createSnp()
}

func createSnp(){
```

```
        self.backView.snp.makeConstraints {(make)in
          make.left.right.equalToSuperview().inset(10)
          make.top.bottom.equalToSuperview().inset(10)
        }

        self.nameLabel.snp.makeConstraints{(make)in
          make.center.equalToSuperview()
      }
    }
```

接下来通过数据源来控制 Cell 上按钮文字的代码：

```
        func update(_object:HomeItem){
    self.nameLabel.text=object.name
        }
```

最后就是单击时的放大效果，其实 Cell 本身自带了单击高亮效果，前面已经通过 selectionStyle 属性将其设置为 UITableViewCellSelectionStyle.none，去除了自带的高亮效果。如何自定义点中 Cell 时的效果呢？其实我们只要重写 UITableViewCell 的 setHighlighted 方法，在其中处理 highlighted 状态时的样式即可。

```
    override func setHighlighted(_highlighted:Bool,animated:Bool){
      super.setHighlighted(highlighted,animated:animated)
      if self.isHighlighted{
        self.backView.transform=self.backView.transform.scaledBy(x:1.1,
y:1.1)
      }else{
       UIView.animate(withDuration:0.5){()->Void in
        self.backView.transform=.identity
       }
      }
    }
```

代码分析：当处于 highlighted 状态时，通过改变 UIView 的 transform 实现 backView 的放大效果，当取消 highlighted 状态时，我们通过 self.backView.transform = .identity 将 backView 的 transform 还原即可，并通过 UIView.animate 添加了一个动画效果，让其还原时的效果不会那么生硬。

完成了 HomeItem、HomeTableViewCell 后，我们就可以完成 HomeViewController 界面了。前面已经实现了 UITableView 的创建和代理方法，接下来就是将数据源和 HomeTableViewCell 进行关联了。

首先就是在 HomeViewController 里添加数据源变量，并实现数据源数据的添加。

```
        var cellArray=[HomeItem]()

        func createData(){
    self.cellArray.append(HomeItem(name:"双人对战",key:"outline"))
```

```
    self.cellArray.append(HomeItem(name:"人机对战",key:"AI"))
    self.cellArray.append(HomeItem(name:"联网对战",key:"online"))
        }
```

关联数据源到 HomeTableViewCell，分别在相关代理方法里加入相关代码。

numberOfRowsInSection 控制 Cell 的数量：

```
    return self.cellArray.count
```

cellForRowAt 方法返回相关 Cell：

```
    let object=self.cellArray[indexPath.row]
    let  cell=HomeTableViewCell(style:UITableViewCellStyle.default,reuseIdenti
fier:"cell")
    cell.update(object)
    return cell
```

在 heightForRowAt 里添加返回 Cell 的高度：

```
    return 50
```

完成 HomeViewController 界面的搭建，即单击界面的 Cell 进行跳转的功能，接下来我们在 Composition 的 Controller 目录下新建 CompositionViewController 控制器，并在 HomeViewController 里 TableView 的代理方法 didSelectRowAt 中实现单击跳转功能，代码如下。

```
    let object=self.cellArray[indexPath.row]
        let to_vc=CompositionViewController()

        //根据 key 值判断不同类型的游戏方式，后期做不同的处理
        switch object.key{
        case"outline":

          break
        case"AI":

          break
        case"online":

          break
        default:
          break
        }
        self.present(to_vc,animated:true,completion:nil)
```

代码分析：首先通过单击 Cell 对应的数据源的 key 值来判断当前的游戏方式，后期会在 CompositionViewController 里添加相应字段进行区分。最后使用 UIViewController 方法里的 present 方法进行界面的跳转。

2．优化代码（一些简单的封装）

看到这里，不知道读者是否会诧异，怎么才开始搭建界面就要优化代码呢？而且是否

也觉得没有什么可优化的？其实越是项目前期越应该注重代码的优化，因为前面不处理，后面等到发现不得不再去优化时，其实已经说明代码有很多地方要修改了，如代码的重用性不够好，一段代码很多地方重复出现等，这就需要我们花费更多的时间去进行修改了。

那么我们优化前面的哪些代码呢？可以先看下前面我们在 HomeTableViewCell 里定义了 nameLabel 的字体颜色时用了这样的代码：

```
UIColor.init(red:0.3,green:0.3,blue:0.3,alpha:1)
```

系统提供的颜色实例化方法其实并不友好，要写很长的一段代码。另外，最主要的还是在实际项目中，设计师在给设计图时都是“#33ff00”这样的 6 位 16 进制的 RGB 格式来表示颜色值的，也可能是加上透明度的 8 位 ARGB 格式。这对开发过程来说就是一个痛点，每次用到颜色时都要去转化一下，浪费很多时间。

因此，我们可以对系统提供的 UIColor 类进行一次扩展，为其实现相应的功能，让我们可以直接使用“33ff00”这样的格式进行实例化。还记得我们前面有的目录结构中有一个 Common 模块，其下有一个 Extension 子文件夹，现在就在这里面创建一个 Extension+UIColor.swift 文件，如图 24-13 所示。

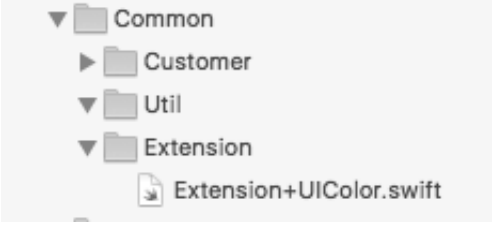


图 24-13

首先我们可以在 extension 里使用 convenience 关键字扩展一个实例化方法：

```
extension UIColor{
        convenience init(_code:String){

        }
}
```

正常情况下，参数 code 的格式可以是“AARRGGBB”的 8 位字符串，如“33445566”。也可以是“RRGGBB”的 6 位字符串，这时默认透明度为 1.0，如“346677”。

对于类似“33445566”这样的格式，也可以简写成“ARGB”的 4 位字符串“3456”。当然透明度为 1.0 时，也可以简写成“RGB”的 3 位字符串“456”。

这里对 code 进行截取字符串，然后将 16 进制转成 10 进制，然后除以 255.0 得到 UIColor 系统提供的实例化方法 init(red: CGFloat, green: CGFloat, blue: CGFloat, alpha: CGFloat)对应的参数。

```
    convenience init(_code:String){
  //首先定义一个 16 进制转化成系统自带颜色值的方法
  let colorComponent={(start:Int,length:Int)->CGFloat in
```

```
        let i=code.index(code.startIndex,offsetBy:start)
        let j=code.index(code.startIndex,offsetBy:start+length)

        //这里 subHex 的类型是 Substring
        let subHex=code[i..<j]

        //如果只有一位数，则代表重复的两位简写
        let subHexStr=length < 2 ? "\(subHex)\(subHex)":String(subHex)

                //16 进制转换为 10 进制
        var component:UInt32=0
        Scanner(string:subHexStr).scanHexInt32(&component)
        return CGFloat(component)/255.0
    }

    //区分字符串的长度判断当前颜色值的格式
    let argb={()->(CGFloat,CGFloat,CGFloat,CGFloat)in
        switch(code.count){
        case 3://#RGB
            let red=colorComponent(0,1)
            let green=colorComponent(1,1)
            let blue=colorComponent(2,1)
            return(red,green,blue,1.0)
        case 4://#ARGB
            let alpha=colorComponent(0,1)
            let red=colorComponent(1,1)
            let green=colorComponent(2,1)
            let blue=colorComponent(3,1)
            return(red,green,blue,alpha)
        case 6://#RRGGBB
            let red=colorComponent(0,2)
            let green=colorComponent(2,2)
            let blue=colorComponent(4,2)
            return(red,green,blue,1.0)
        case 8://#AARRGGBB
            let alpha=colorComponent(0,2)
            let red=colorComponent(2,2)
            let green=colorComponent(4,2)
            let blue=colorComponent(6,2)
            return(red,green,blue,alpha)
        default:
            return(1.0,1.0,1.0,1.0)
```

```
        }}

        let color=argb()
        self.init(red: color.0,green:color.1,blue:color.2,alpha:color.3)
    }
```

对于以上代码这里就不再详细讲解了，代码中有相应的注释。这样的代码就已经可以作为一个工具了，在以后的项目中都能用到。

我们可以优化相关代码了，将 HomeTableViewCell 中的 nameLabel 字体颜色代码改为：

```
self.nameLabel.textColor=UIColor("333")
```

这样的代码看着也更加清晰了。但在正常情况下，项目都会去对文字的颜色有着统一的设计，而且也有可能在项目迭代过程中对字体颜色进行统一更改。那么在项目前期如何更好地处理这样的需求呢？其实我们只要定义一个全局的常量来对颜色进行统一管理即可。

在 Swift 中，全局常量一般怎么定义呢？使用 struct，并使用 static let 来定义。接下来我们就在 General 目录下新建一个子目录 Marco，然后创建一个 Color.swift 文件，如图 24-14 所示。

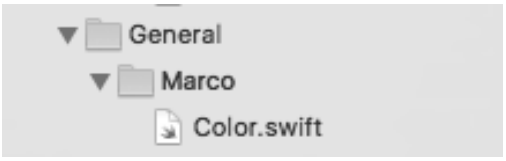


图 24-14

创建一个 Color 的 struct，并在其中添加相应的系统统一颜色。

```
struct Color{
        //字体色系
        static let textH=UIColor("333333")
        static let textM=UIColor("666666")
        static let textL=UIColor("999999")

        //背景色
        static let mainBg=UIColor("d0d0d0")
}
```

将 nameLabel 字体颜色代码改为：

```
self.nameLabel.textColor=Color.textH
```

将 BaseViewController 中设置背景色的代码改为：

```
self.view.backgroundColor=Color.mainBg
```

同理，我们还可以创建一个字体大小样式的 struct。

```
import UIKit
```

```
struct Font{
        //字体大小
        static let small=UIFont.systemFont(ofSize:13)
        static let middle=UIFont.systemFont(ofSize:15)
        static let large=UIFont.systemFont(ofSize:17)

}
```

将 HomeTableViewCell 中的 nameLabel 字体大小代码改为：

```
self.nameLabel.font=Font.middle
```

总之，在项目开发过程中，适当地停下来，审视之前的代码，对代码进行一定的优化，会为项目后期开发减少更多的时间，而且一些简单的封装也可以直接用到其他项目中去。随着项目经验的不断积累，之后开发项目的效率就会大大提高。

3．CompositionViewController 控制器的界面框架搭建

先来看看游戏界面的“庐山真面目”吧，如图 24-15 所示（此图只是暂时的）。

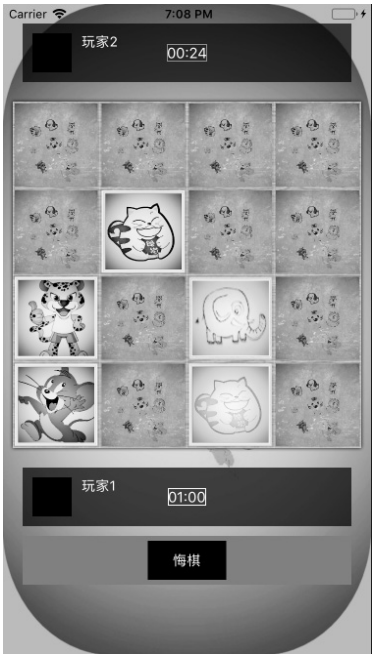


图 24-15

从图中可以看出，该界面分为玩家 1、玩家 2、棋局、相关设置四个视图部分。玩家的视图上有头像和名称，棋盘的视图上放着棋子。设置的视图有返回、悔棋等按钮。

首先将当前界面的这四部分的布局框架搭建好，初始化这四个 View，在 CompositionViewController 控制器里加入以下代码：

```
lazy var playerOneView:UIView={
  let v=UIView()
```

```
    return v
}()
lazy var playerTwoView:UIView={
    let v=UIView()
    return v
}()
lazy var setView:UIView={
    let v=UIView()
    return v
}()
lazy var compositionView:UIView={
    let v=UIView()
    return v
}()
```

暂时先将视图定义为 UIView，后面再对应地加入相关自定义的视图。接下来将视图加到控制器主视图中。

```
        func createView(){
    self.view.addSubview(playerOneView)
    self.view.addSubview(playerTwoView)
    self.view.addSubview(setView)
    self.view.addSubview(compositionView)
}
```

使用 SnapKit 进行布局。

```
    func createSnp(){
  self.playerTwoView.snp.makeConstraints{(make)in
    make.top.equalToSuperview().inset(20)
    make.left.right.equalToSuperview().inset(20)
    make.height.equalTo(60)
  }

  self.playerOneView.snp.makeConstraints{(make) in
    make.top.equalTo(self.compositionView.snp.bottom).offset(10)
    make.left.right.equalToSuperview().inset(20)
    make.height.equalTo(60)
  }

  self.compositionView.snp.makeConstraints{(make)in
make.top.equalTo(self.playerTwoView.snp.bottom).offset(10)
    make.left.right.equalToSuperview()
    make.height.equalTo(self.compositionView.snp.width)
  }
```

```
    self.setView.snp.makeConstraints{(make)in
      make.top.equalTo(self.playerOneView.snp.bottom).offset(10)
      make.left.right.equalToSuperview().inset(20)
      make.height.equalTo(50)
    }
  }
```

在 viewDidLoad 方法里调用 createView 和 createSnp 方法。

```
override func viewDidLoad(){
super.viewDidLoad()
self.createView()
self.createSnp()
}
```

运行项目，进入 CompositionViewController 界面，如图 24-16 所示。完成了界面的基本布局后，对四个子视图进行具体布局。

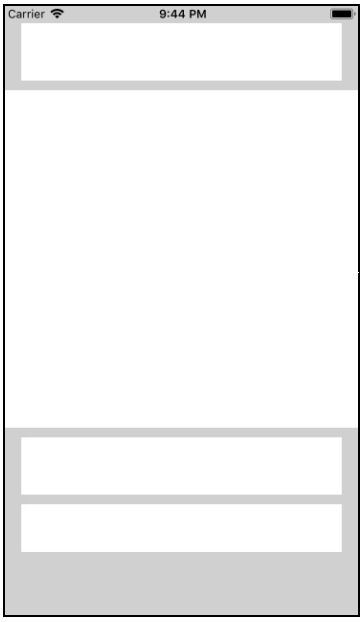


图 24-16

4．PlayerView 视图的搭建

玩家视图的控件搭建，在 Composition 模块的 View 文件夹下创建一个继承于 UIView 的 PlayerView 类文件。玩家视图内有两个子控件：头像和名称。先完成头像和名称的创建。

```
  import UIKit

  class PlayerView:UIView{
    var avatarImageView:UIImageView!
```

```
    var nameLabel:UILabel!

    convenienceinit(){
      self.init(frame:CGRect.zero)
    }
    override init(frame:CGRect){
      super.init(frame:frame)
      self.createView()
    }

    required init?(coder aDecoder:NSCoder){
      fatalError("init(coder:)has not been implemented")
    }

    func createView(){
      self.avatarImageView=UIImageView()
      self.avatarImageView.backgroundColor=Color.mainBg
      self.addSubview(self.avatarImageView)

      self.nameLabel=UILabel()
      self.nameLabel.textColor=Color.textH
      self.nameLabel.font=Font.middle
      self.nameLabel.text="玩家昵称"
      self.addSubview(self.nameLabel)
  }
}
```

接下来就是布局约束了。观察一下，玩家 1 和玩家 2 的布局是有一定区别的，玩家 1 的头像和名称在视图右边，玩家 2 的在视图左边，那么这里的布局就要做不同的处理了。

```
        //头像在左边的约束
func snpLeft(){
  self.avatarImageView.snp.makeConstraints{(make)in
    make.top.left.bottom.equalToSuperview().inset(10)
    make.height.equalTo(self.avatarImageView.snp.width)
  }
  self.nameLabel.snp.makeConstraints{(make)in
    make.left.equalTo(self.avatarImageView.snp.right).offset(-10)
    make.top.equalTo(self.avatarImageView)
  }
}

//头像在右边的约束
func snpRight(){
  self.avatarImageView.snp.makeConstraints{(make)in
```

```
        make.top.right.bottom.equalToSuperview().inset(10)
        make.height.equalTo(self.avatarImageView.snp.width)
      }
      self.nameLabel.snp.makeConstraints{(make)in
        make.right.equalTo(self.avatarImageView.snp.left).offset(10)
        make.top.equalTo(self.avatarImageView)
      }
    }
```

将控制器中的玩家视图换成 PlayerView，并对玩家 1 和玩家 2 使用不同的布局约束。

```
    lazy var playerOneView:PlayerView={
let v=PlayerView()
v.snpRight()
v.backgroundColor=UIColor.white
return v
    }()
    lazy var playerTwoView:PlayerView={
let v=PlayerView()
v.snpLeft()
v.backgroundColor=UIColor.white
return v
    }()
```

运行后得到的界面如图 24-17 所示。

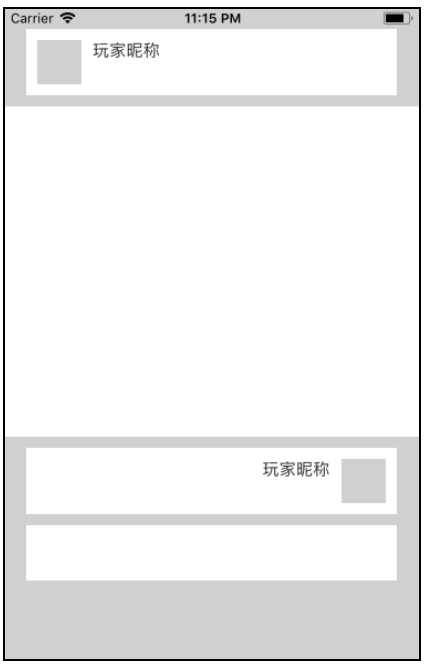


图 24-17

5．SetView 视图的搭建

下面搭建最下方的子视图设置视图 SetView。这部分只有三个按钮，即左边的返回按钮，中间的悔棋按钮，右边的认输按钮。使用自定义视图的方式，在 View 目录下新建一个 SetView.swfit 文件，同样 SetView 继承于 UIView，然后就是完成其内三个控件的创建和布局了。

```
import UIKit

class SetView:UIView{
  var backButton:UIButton!
  var regretButton:UIButton!
  var surrenderButton:UIButton!

  override init(frame:CGRect) {
    super.init(frame:frame)
    self.createView()
    self.createSnp()
  }

  required init?(coder aDecoder:NSCoder){
    fatalError("init(coder:)has not been implemented")
  }

  func createView(){
    self.backButton=UIButton()
    self.backButton.setTitle("返回",for:.normal)
    self.backButton.setTitleColor(Color.textH,for:.normal)
    self.addSubview(self.backButton)

    self.regretButton=UIButton()
    self.regretButton.setTitle("悔棋",for:.normal)
    self.regretButton.setTitleColor(Color.textH,for:.normal)
    self.addSubview(self.regretButton)

    self.surrenderButton=UIButton()
    self.surrenderButton.setTitle("认输",for:.normal)
    self.surrenderButton.setTitleColor(Color.textH,for:.normal)
    self.addSubview(self.surrenderButton)

  }

  func createSnp(){
```

```
        self.backButton.snp.makeConstraints{(make)in
          make.top.bottom.equalToSuperview().inset(5)
          make.left.equalToSuperview().inset(10)
        }

        self.regretButton.snp.makeConstraints{(make)in
          make.top.bottom.equalToSuperview().inset(5)
          make.center.equalToSuperview()
        }

        self.surrenderButton.snp.makeConstraints{(make)in
          make.top.bottom.equalToSuperview().inset(5)
          make.right.equalToSuperview().inset(10)
        }
      }
    }
```

在控制器 CompositionViewController 里修改 setView 的类型为 SetView：

```
    lazy var setView:SetView={
let v=SetView()
v.backgroundColor=UIColor.white
return v
    }()
```

6．CompositionView 视图的搭建

最后剩下的就是棋局部分的视图搭建了，这里其实包括了两部分：棋盘和棋子。先创建 CompositionView 这样的一个棋局容器。

```
import UIKit
//棋局(棋盘+棋子)
class CompositionView:UIView{

  override init(frame:CGRect){
    super.init(frame:frame)
    self.createView()
  }

convenience init(){
    self.init(frame:CGRect.zero)

  }

  required init?(coder aDecoder:NSCoder){
    fatalError("init(coder:)has not been implemented")
```

```
    }

    func createView(){

    }
}
```

完成棋局容器文件的创建，也就是完成其中棋盘和棋子的创建。先来看看如何搭建棋盘，棋盘也就是一个 4×4 的方格视图，包括一张图和方格之间的分割线。所以我们可以使用 UIImageView 控件并在其上画出相应的分割线，如何画分割线呢？我们可以重写 UIView 的 draw(_ rect: CGRect)方法，这个方法就是在 View 视图绘制时调用的。这里需要注意，不能直接重写 UIImageView 的 draw 方法，苹果官方文档指出 UIImageView 是专门显示图片的控件，用了最优显示技术，不让调用 darwrect 方法。所以正确的方法是定义一个类继承一个 UIView，在其上绘制分割线，并将这个视图加到 UIImageView 上。

```
import UIKit

class ChessBoardLine: UIView {

  override init(frame: CGRect) {
    super.init(frame: frame)
  }

convenience init(){
  self.init(frame: CGRect.zero)
}

required init?(coder aDecoder: NSCoder) {
  fatalError("init(coder:) has not been implemented")
}

var lineColor: UIColor = UIColor.white{
  didSet{
   self.setNeedsDisplay()
  }
}
override func draw(_ rect: CGRect) {
  super.draw(rect)
  //划分割线
  guard let context = UIGraphicsGetCurrentContext() else {
    return
}
context.setStrokeColor(lineColor.cgColor)
```

```
        context.setLineWidth(1)

        //循环加入 3 条横向分割线
        for i in 0..<3{
          let y = CGFloat(i+1) * rect.height/4
          context.move(to: CGPoint(x: 0, y: y))
          context.addLine(to: CGPoint(x: rect.width, y: y))
          context.strokePath()
        }
        //循环加入 3 条纵向分割线
        for i in 0..<3{
          let x = CGFloat(i+1) * rect.width/4
          context.move(to: CGPoint(x: x, y: 0))
          context.addLine(to: CGPoint(x: x, y: rect.height))
          context.strokePath()
        }
      }
    }
```

使用 UIGraphicsGetCurrentContext()进行画线，线条颜色可以设置，改变线条颜色时调用 self.setNeedsDisplay()方法，此时会重新绘制视图。下面将线条加到棋盘视图上。

```
    import UIKit

    class ChessBoard: UIImageView {

      override init(frame: CGRect) {
        super.init(frame: frame)
        self.createView()
      }

      convenience init(){
        self.init(frame: CGRect.zero)
      }

      required init?(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
      }

      //设置棋盘分割线颜色
      var lineColor: UIColor = UIColor.white{
        didSet{
          self.lineView?.lineColor = lineColor
          self.lineView?.layer.borderColor = lineColor.cgColor
```

```
    }
  }
  private var lineView: ChessBoardLine!

  override func layoutSubviews() {
    super.layoutSubviews()
    self.lineView?.frame = self.bounds
  }

  func createView(){

    self.layer.shadowColor = UIColor.black.cgColor
    self.layer.shadowOffset = CGSize(width: 0, height: 0)
    self.layer.shadowRadius = 2
    self.layer.shadowOpacity = 2

    self.isUserInteractionEnabled = true

    self.lineView = ChessBoardLine()
    self.lineView.backgroundColor = UIColor.clear
    self.lineView?.layer.borderWidth = 1.0
    self.addSubview(self.lineView)

    self.lineColor = UIColor.white.withAlphaComponent(0.3)
  }

}
```

棋子在走棋过程中是被单击的，所以可以将其看成一个按钮，那么我们可以定义一个 ChessButton 并继承于 UIButton。

```
import UIKit

class ChessButton: UIButton {

  override init(frame: CGRect) {
    super.init(frame: frame)
    self.createView()
  }

  convenience init() {
    self.init(frame: CGRect.zero)

  }
```

```
    required init?(coder aDecoder: NSCoder) {
      fatalError("init(coder:) has not been implemented")
    }

    //一些界面的初始设置
    func createView(){

    }
  }
```

棋子分为正、反两面，开局时所有棋子背面朝上，单击后棋子翻转过来，其实背面和正面都是图片，也就是两个 UIImageView，这里为了方便学习，我们暂时使用背景色和文字来表示棋子。然后调用系统提供的翻转效果方法。

```
UIView.transition(from fromView: UIView, to toView: UIView, duration: TimeInterval, options: UIViewAnimationOptions = [], completion: ((Bool) -> Swift.Void)? = nil)
```

改变方法实现翻转动画效果，并会将 fromView 从父视图中移除，将 toView 加入 fromView 的父视图中，duration 设置动画效果时长，options 设置翻转的方向，并在完成后调用相应的闭包方法。

这样我们就可以定义两个 UIImageView 在 ChessButton 中，并封装一个翻棋方法了。

```
    var chessImageView: UIImageView!
    var backImageView: UIImageView!
    var chessLabel: UILabel!

    func createView(){
    //先将背面视图加入
    self.backImageView = UIImageView()
    self.backImageView.backgroundColor = UIColor.black
    self.addSubview(self.backImageView)

    self.chessImageView = UIImageView()

    self.chessLabel = UILabel()
    self.chessLabel.font = UIFont.boldSystemFont(ofSize: 30)
    self.chessLabel.textColor = UIColor.white
    self.chessLabel.textAlignment = .center

    self.chessImageView.addSubview(self.chessLabel)

  }

  func update() {
    self.chessLabel.text = "棋"
```

```
      self.chessLabel.backgroundColor = UIColor.red
    }

      //翻转
    func turnChess(_ complish: @escaping ()->()){
      UIView.transition(from:  self.backImageView,  to:  self.chessImageView,
duration:1.0,options:UIViewAnimationOptions.transitionFlipFromLeft){(finished)
in
        if finished{
          complish()
        }
      }
    }
```

这里因为棋子里面的视图大小和棋子本身的大小是一致的，且控件并不是固定在棋子内的子视图的（开始只是背面视图，翻转后只是正面视图），我们可以通过重写 layoutSubviews 方法，在其中设置子视图的 frame。

```
        override func layoutSubviews() {
        super.layoutSubviews()
        self.chessImageView.frame = self.bounds
        self.backImageView.frame = self.bounds
        self.chessLabel.frame = self.bounds
    }
```

有了棋盘和棋子，接下来就是完善棋局容器 CompositionView 视图了。开局时，棋局容器也就是由 1 个棋盘和 16 个棋子组成。

```
    class CompositionView: UIView {

      //棋盘
      var chessboard: ChessBoard!
      var chessButtons = [ChessButton?]()

      var wh: CGFloat = 0
      var padding: CGFloat  = 2.0

      override init(frame: CGRect) {
        super.init(frame: frame)
        self.createView()
        self.createSnp()
      }

      convenience init() {
        self.init(frame: CGRect.zero)
```

```
    }

    required init?(coder aDecoder: NSCoder) {
      fatalError("init(coder:) has not been implemented")
    }

    func createView(){
      //棋盘暂时用绿色背景代替图片
      self.chessboard = ChessBoard()
      self.chessboard.backgroundColor = UIColor.green
      self.addSubview(self.chessboard)

    }

    func createSnp(){
      self.chessboard.snp.makeConstraints { (make) in
        make.edges.equalToSuperview().inset(10)
      }
    }

    override func layoutSubviews() {
      super.layoutSubviews()
      self.wh = self.chessboard.frame.width/4

      self.loadChess()
    }

    func loadChess(){
      //测试数据
      for _ in 0...15{
        let chessBtn = ChessButton()
        chessBtn.addTarget(self,action:#selector(self.didChess(_:)),for:.
touchUpInside)
        self.chessButtons.append(chessBtn)
      }

      for (i, btn)in self.chessButtons.enumerated(){
        if btn == nil{
          continue
      }

      let x = CGFloat(i%4)*wh
```

```
        let y = CGFloat(i/4)*wh
        btn?.frame = CGRect(x: x+padding, y: y+padding, width: wh-padding*2,
height: wh-padding*2)
        self.chessboard.addSubview(btn!)
      }
    }

    @objc func didChess(_ chessBtn: ChessButton){
      chessBtn.turnChess {

      }
    }

    }
```

分析上面的代码，我们使用[ChessButton?]一个数组来存放棋子控件，并通过 loadChess 方法，加入测试数据，循环棋子数组，分别设置棋子的 frame，并将其加入棋子视图。这里为何要用可选类型呢？这是因为棋子可能存在被吃了的情况。棋子被吃相应的位置应是空，而不是改变数组的长度，这里数组的长度应始终为 16。

接下来就是将控制器中的 compositionView 进行替换：

```
        lazy var compositionView: CompositionView = {
        let v = CompositionView()
        v.backgroundColor = UIColor.white
        return v
    }()
```

运行项目测试的效果如图 24-18 所示。

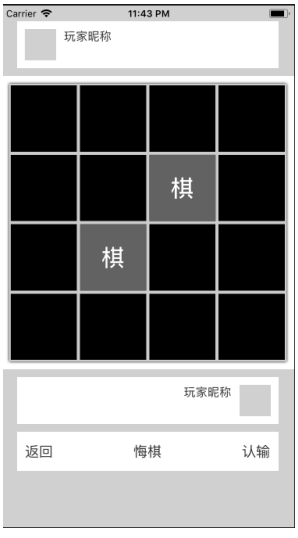


图 24-18

24.4 相关 model 和走棋逻辑的实现

所谓 model，顾名思义，就是模型，即对现实世界中的物体用一个数据模型表达出来。前面已经将基本的界面布局搭建完成，接下来就是通过 model 将数据填充到界面中去了。

根据前面界面布局代码可以看出，本案例所涉及的模型也就是棋子、棋盘、玩家、棋局、走棋记录等。

1．棋子 model 的创建

下面开始创建第一个 model（棋子）。首先分析如何用数据模型来构建棋子。前面介绍的游戏规则是，游戏分为有 2 方，8 种动物，共 16 张卡牌。也就是说，棋子有个属性，代表棋子属于哪一方，同时还有一个属性，代表棋子是什么动物。同时游戏中棋子有的是非被翻开的状态，那么就是有个属性来代表棋子当前的状态。

实际代码中用什么来表示所属方，以及是否直接用动物的名称代表棋子是什么动物吗？棋子与棋子之间又有着怎样的关系呢？先要想想棋子在游戏中有怎样的业务逻辑。首先，按照一定的顺序和规则来判断当前棋子是否可以吃掉对方棋子。棋子本身有一个方法来判断是否可以吃掉另一个棋子。

基于上述分析，我们可以:

（1）用 0 和 1 代表棋子所属方；

（2）用 1～8 这 8 个数字代表棋子是什么动物，1 代表象、2 代表狮、…、8 代表鼠；

（3）编写一个判断棋子是否可以吃掉另一枚棋子的方法。

接下来创建 Swift 文件，棋子的 model：Chess，并完成以下代码，完成棋子数据模型的构建。

```
struct Chess {
        var belong: Int //棋子的所属方（0、1）
        var value: Int //棋子的值（1～8）
        var isShow: Bool = false  // 当前棋子是否翻开，默认为没有翻开

        //初始化方法
        init(belong: Int, value: Int){
        self.belong = belong
        self.value = value
        }

        //判断是否可以吃子
        func canEat(_ chess: Chess)->Bool{

        }
}
```

通过上面的代码，我们发现，判断是否可以吃子的方法并没有具体实现。那么如何去实现呢？方法来自游戏规则。我们要做的就是通过游戏规则的分析来实现这个方法。

下面开始分析，可以吃子就是棋子的值（value）属性比另一个棋子的小，同时排除最后一个棋子（鼠，value=8）可以吃第一个棋子（象，value=1）的情况，同时也别忘了是同一方的棋子也不能吃，即 belong 属性相等的情况。

经过分析，可能会有一些读者写出类似下面的代码：

```
     func canEat(_ chess: Chess)->Bool{
    if self.belong != chess.belong{
      if  self.value <= chess.value{
        if !(self.value == 8 && chess.value == 1){
          return true
        }
      }
    }
    return false
}
```

如果案例中我们也给出这样的代码，那么就违背了我们写这个案例的初衷。我们希望实战案例带来的不仅是语法，更多的是思考分析问题的方式，以及写出质量更高的代码。先看看上述代码的缺点，显而易见，缺点就是 if 语句嵌套层次过多，可读性差。

如何进行改造呢？就是要把逻辑更清晰地表达出来，让代码有一定的顺序性，并能够一目了然。这样代码的可读性就大大增强了。首先我们对前面的分析结果给出一定的顺序。例如，如果双方棋子相同那么就直接反回（return）不可以吃（false），不需要再判断棋子的大小了。再进一步处理象和鼠的特殊关系，最后再判断当前棋子和另一个棋子值的大小。最终代码如下。

```
func canEat(_ chess: Chess)->Bool{
     //如果棋子还没有翻开，则不可吃
    if !chess.isShow{
      return false
    }
       //如果是同一方棋子，则不可吃
    if self.belong == chess.belong{
      return false
    }
    //如果当前棋子是 8（鼠），则可以吃 1（象）
    if self.value == 8 && chess.value == 1{
      return true
    }
       //如果当前棋子是 1（象），则不可以吃 8（鼠）
    if self.value == 1 && chess.value == 8{
      return false
```

```
        }
        //否则，如果前棋子的值小于或等于的另一个棋子，则可以吃
        return  self.value <= chess.value
      }
```

为何这么处理呢？其实在比较老的编程语言或比较古板的技术公司里，会要求函数遵循单一出口原则，即一个函数体里就在最后有一个 return 代码。但对于 Swift 这样的比较先进的语言来说并不是这样的，更加推崇的还是尽早返回（return）一个函数，这点从 Swift 的关键字 guard 就可以看出一二。

接下来就是设置棋子的名称了。棋子的名称是根据棋子的值来设置的。这样我们可以定义一个只读字段，通过棋子的值返回对应的棋子名称。

因为棋子的名称是固定的，这样我们就定义一个全局常量来存放棋子的名称。我们在 General 的 Marco 目录下新建文件 Define.swift。

```
struct Define{
        static let chessNames = ["象","狮","虎","豹","狼","狗","猫","鼠"]
}
```

在 Chess 的 model 里声明只读属性 name，并根据 value 返回对应的名称：

```
var name: String{
 return Define.chessNames[value-1]
}
```

有了棋子 model，接下来我们就可以将 model 和视图 ChessButton 关联起来了，可以在 ChessButton 里声明一个 chess 属性，并对其实例化方法进行相应修改。同时将具体棋子 name 对应到 chessLabel 上，根据棋子的所属方，设置不同的两种背景色。

```
    var chess:Chess!

    convenience init(_chess:Chess){
self.init(frame:CGRect.zero)
self.chess=chess
    self.update()
    }

    func update(){
self.chessLabel.text=self.chess.name
self.chessLabel.backgroundColor = self.chess.belong == 0 ? UIColor.red:
UIColor.blue
    }
```

紧接着修改 CompositionView 里 loadChess 方法添加的测试棋子数据。这样就完成了棋子的模型和视图的关联。

```
//测试数据
    for i in 1...8{
      let chessBtn=ChessButton(Chess(belong:0,value:i))
```

```
          chessBtn.addTarget(self,action:#selector(self.didChess(_:)),for:.
touchUpInside)
          self.chessButtons.append(chessBtn)
        }

        for i in 1...8{
          let chessBtn=ChessButton(Chess(belong:1,value:i))
          chessBtn.addTarget(self,action:#selector(self.didChess(_:)),for:.
touchUpInside)
          self.chessButtons.append(chessBtn)
        }
```

2．棋盘的创建

其实棋盘的数据模型也是很简单的，棋盘本身由 4×4 的 16 个方格组成。这样我们可以用一个二维坐标系来表示一个棋盘，其中相应的坐标表示棋盘的位置，如图 24-19 所示。

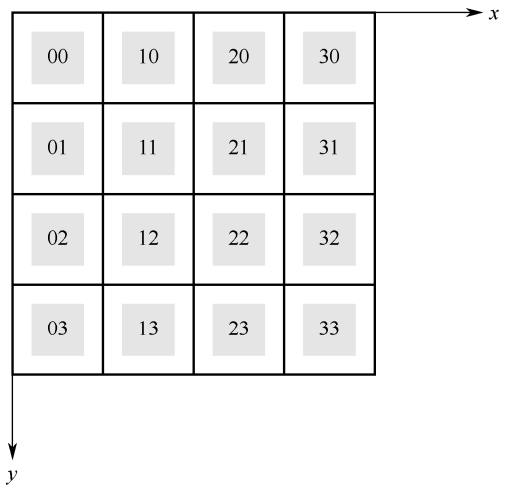


图 24-19

棋子是放在棋盘上的，为了更好地表示棋子在棋盘的位置，我们可以先定义一个 struct 来表示棋子的位置。

```
    struct CBPosition{
        var x: Int
        var y: Int
    }
```

创建一个 Chessboard.swfit 文件，定义一个棋盘 model。因为我们这里固定了棋盘坐标的范围，所以可以声明两个常量来表示 *x* 坐标和 *y* 坐标的最大值。棋子走棋时需要根据棋盘的坐标范围来判断是否可以向某个方向走棋，可以定义一个方法来判断坐标是否在棋盘内。

```
struct Chessboard {
        let width: Int = 4
        let height: Int = 4

        //判断位置是否在棋盘内
        func isInBoard(by position: CBPosition)->Bool{
        return (0..<self.width).contains(position.x) &&
(0..<self.height).contains(position.y)
        }
}
```

给棋子声明一个位置属性，代表棋子在棋盘中的位置。

3．自定义运算符实现棋子在棋盘中的走棋逻辑

棋子的走棋，其实就是改变其在棋盘中的位置，走棋方向也就是上、下、左、右，每次移动一格。这样我们可以使用 postfix operator 自定义 4 个单目运算符，以计算坐标移动后的位置。创建 Moveable.swift 文件，自定义运算符和走棋协议。

```
postfix operator >+ //右移
postfix operator >- //左移
postfix operator ^+ //下移
postfix operator ^- //上移

protocol Moveable{
        static postfix func >+ (left: Self) -> Self
        static postfix func >- (left: Self) ->Self
        static postfix func ^+ (left: Self) -> Self
        static postfix func ^- (left: Self) -> Self
}
```

让 CBPosition 实现 Moveable 协议，并实现对应移动的方法。

```
extension CBPosition: Moveable{

        //右移，相当于 x 坐标加 1
        static postfix func >+(left: CBPosition) -> CBPosition {
        return CBPosition(x: left.x+1, y: left.y)
        }

        //左移，相当于 x 坐标减 1
        static postfix func >-(left: CBPosition) -> CBPosition {
        return CBPosition(x: left.x-1, y: left.y)
        }
```

```
        //上移，相当于 y 坐标减 1（坐标系 y 轴方向朝下）
        static postfix func ^+(left: CBPosition) -> CBPosition {
        return CBPosition(x: left.x, y: left.y-1)
        }

        //下移，相当于 y 坐标加 1（坐标系 y 轴方向朝下）
        static postfix func ^-(left: CBPosition) -> CBPosition {
        return CBPosition(x: left.x, y: left.y+1)
        }
}
```

最后给棋子声明位置属性，以记录棋子当前的位置。在 Chess.swift 文件里加入代码：

```
var position: CBPosition?
```

4．走棋记录 model 的创建

走棋记录就是把走棋数据记录下来吗？这其实关系到另一个功能，那就是悔棋。想要悔棋，就要知道前面走了哪一步。而且，后期还要实现联网对战功能，如果在网络不稳定、出现传输数据丢失的情况，走棋记录进而也可以保证棋局数据的正确性。

走棋记录其实也需要一个数据模型。走棋记录也就是记录当前某个玩家走了哪一步棋，那么对应的 model 就应有这些相关的属性，即走棋玩家 id、所走的棋子、走棋情况（上、下、左、右走棋，或者翻棋）。走棋情况该用什么来表示呢？因为走棋情况是固定的几种，所以我们可以用一个枚举类型来表示。知道了这些情况，我们就可以开始构造走棋记录的数据模型了。

```
//走棋方向：翻棋、上、下、左、右
enum StepDirection{
        case turn
        case top
        case bottom
        case left
        case right
}

struct Step {
        var step: Int //当前的步数
        var compositionId: String //棋局的 id
        var playerId: String //当前走棋的玩家 id
        var chess: Chess //当前走的棋
        var position: Int //当前走棋的位置
        var direction: StepDirection//走棋的情况
        var beEat: Chess? //被吃的棋子(可能存在)
}
```

5．玩家 model 的创建

接下来就是创建玩家的数据模型了。玩家的数据模型其实也就是玩家的基本信息，即玩家 id、玩家名称、玩家头像等。

```
struct Player {
        var id: String //玩家 id
        var name: String //玩家名称
        var avatar: String //玩家头像
}
```

玩家 model 创建完成后，即可开始和玩家视图关联，这也是很简单的。首先自然能想到的就是，在 PlayerView 里声明一个新的属性：player，属性声明后加上 didSet 方法，让属性被赋值后，自动执行更新相关界面的操作。

```
    var player: Player!{
didSet{
  self.update()
}
    }

    func update(){
self.nameLabel.text = self.player?.name
    暂时用本地图片
self.avatarImageView.image = UIImage(named: self.player!.avatar)
    }
```

6．棋局 model 的创建

有了棋子、棋盘和玩家，就可以开始游戏了。每局游戏，其实也就是一个数据模型，即棋局 model。如何构建这样一个棋局 model 呢？

当游戏开始时，16 个棋子是随机分配到棋盘上的，那么每个棋局就应有这样一个属性，即开局时的棋子分布情况。

这样我们就可以开始创建棋局 Composition 的数据模型了。

```
struct Composition {
        var id: String
        var startChesses: [Chess] //开局棋子布局数据。
        //开局...初始化方法
        init(id: String, startChesses: [Chess]){
        self.id = id
        self.startChesses = startChesses
        }
}
```

至此，完成了游戏中涉及的所有 model 的创建，回顾一下我们创建了哪些文件，这些文件都是在 Section/Composition/Model 目录下的，如图 24-20 所示。

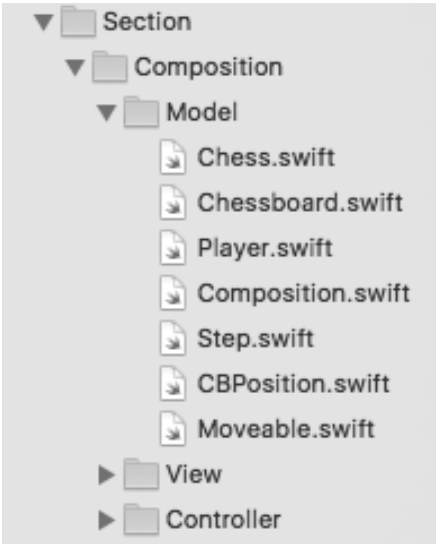


图 24-20

24.5　单机双人对战的实现

数据模型和界面都基本搭建完成了，接下来就是实现游戏的具体功能了，下面先来看看单机版的双人对是战如何实现的。

1．开局数据的创建

当游戏开始时，16 个棋子是随机分布在棋盘中的。我们需要对应地写一个随机分布棋子的算法。我们可以创建一个棋局工具类，来处理一些棋局中的算法。可以在 Compositon 文件夹下新建一个子目录 Manager，并在其中创建一个 CompositionUtil.swift 的类文件，如图 24-21 所示。

如何实现随机生成棋子的算法呢？我们可以先定义一个方法生成 16 个棋子的数组。

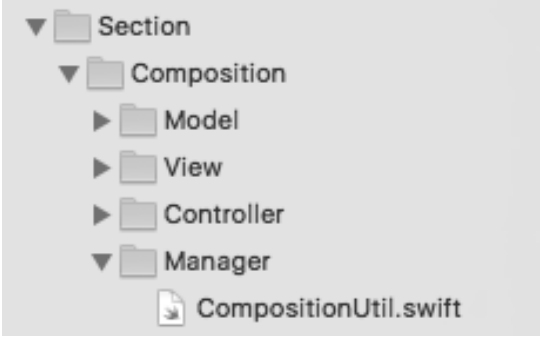


图 24-21

```
class CompositionUtil: NSObject {
        //创建所有棋子
        class func createChess()->[Chess]{
        var chesses = [Chess]()
        for i in 1...8{
        let chess0 = Chess(belong: 0, value: i)
        chesses.append(chess0)
        let chess1 = Chess(belong: 1, value: i)
        chesses.append(chess1)
        }
        return chesses
        }
}
```

按照从左到右、从上到下的顺序依次排列，让数组的下标可以代表棋盘的位置，这样我们就可以通过数组下标给棋子的位置属性赋值了，如图 24-22 所示。

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

图 24-22

我们可以给 CBPosition 加一个实例化方法，让其可以通过一个 0～15 的数字来生成相应的位置坐标。对照一下可知下标除以 4 的余数就是 *x* 坐标，值就是 *y* 坐标。

```
struct CBPosition{
        var x: Int
        var y: Int

        init(x: Int, y: Int){
        self.x = x
        self.y = y
        }

        init(index: Int){
        self.x = index%4
```

```
            self.y = index/4
            }
    }
```

再通过随机数算法，将其顺序打乱，放入一个新的棋子数组内。在 CompositionUtil 里加入一个新的类方法。

```
        //生成随机的棋局
    class func randomQue()->[Chess]{
      var positionArr = [Chess]()
      var ranArr = CompositionUtil.createChess()

      let N = ranArr.count
      for i in 0..<N{
        let index: Int = Int(arc4random() % UInt32(ranArr.count))
        //棋子的位置
        ranArr[index].position = CBPosition(index: i)
        positionArr.append(ranArr[index])
        ranArr.remove(at: index)
      }
      return positionArr
    }
```

2．棋局管理类 CompositionManager 的创建

有了界面，也有了数据模型，接下来就需要将整个游戏过程的业务逻辑完整地体现出来。为了让项目中走棋的业务逻辑和视图分开，我们需要创建一个棋局管理类，来处理游戏中棋局的业务逻辑（游戏规则），并让数据和界面之间关联起来。首先在 Manager 目录下再创建一个类文件：CompositionManager。如何将这个类中的业务逻辑和界面关联起来呢？我们可以通过控制器，在控制器使用代理模式，分别来监听界面的交互和走棋过程，并将相应结果传递给 manager 或 view，如图 24-23 所示。

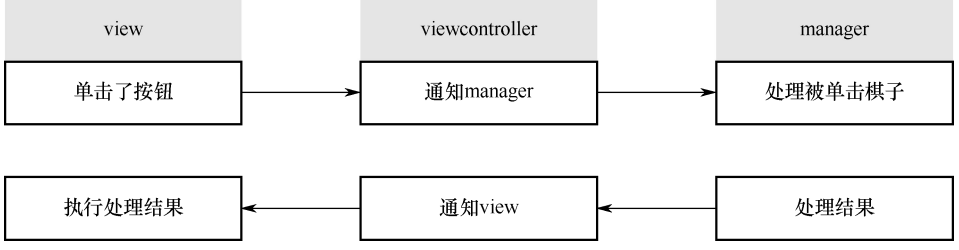


图 24-23

如图 24-23 所示，视图层将单击事件通过控制器告诉 manager，manager 对被单击的棋子进行业务逻辑判断和处理，并将结果通过控制器告知 view，view 对结果做出相应的动作。为了更好地表达整个游戏的过程，我们画了一个整体的走棋流程图，如图 24-24 所示。

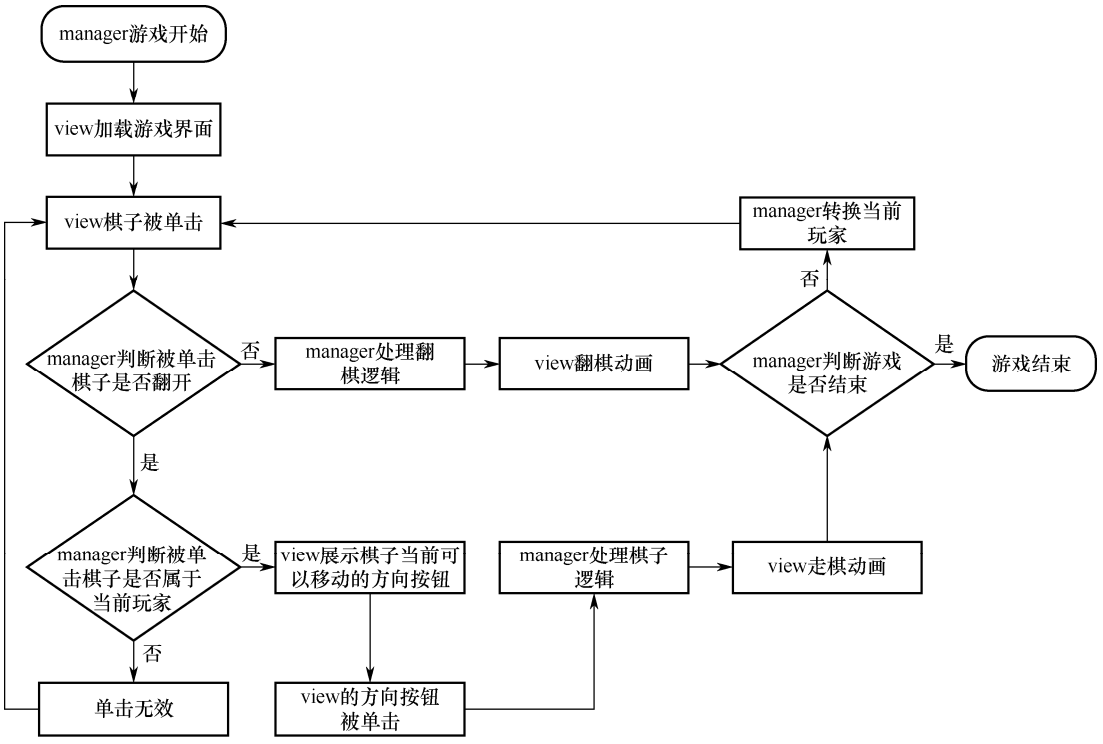


图 24-24

先来看看这个棋局管理类如何处理。既然是棋局管理类，那么就应包含整个游戏过程中的所有元素，那么这里应有哪些元素呢？分析整个游戏过程，开局后，第一个玩家来了，开始翻棋，这时就确定了棋子的所属方，接下来另一个玩家开始翻棋了，对应的就应有玩家 1 和玩家 2 两个属性。之后就是玩家 1 和玩家 2 不断翻棋、走棋的过程，直到一方获胜，游戏结束。所以对应的就有走棋记录、棋子所属方的关联方法、走棋的方法，以及判断游戏是否结束的方法等。首先声明这些基本的属性和方法。

```
class CompositionManager: NSObject {
        var composition: Composition!
        var playerOne: Player! //玩家一
        var playerTwo: Player! //玩家二
        var stepArr = [Step]()//走棋记录的数组。
        var curChesses = [Chess?]()//游戏中记录当前的棋盘上的棋子情况
        var curPlayer: Player!//游戏中记录当前走棋的玩家
}
```

有了这样一个管理类，我们如何开始游戏呢？还记得这个游戏是分几种不同的游戏模式吗？目前我们要实现的是单机双人对战，后面我们还会实现人机对战和联网对战。我们是根据首页中的三个不同的按钮来判断当前的游戏模式的，那么我们在单击了首页选择模式的按钮时就应给棋局加入玩家和开局数据了。首先在 CompositionManager 里定义载入初始数据的方法。

```
    //玩家加入
    func playerJoin(playerOne: Player, playerTwo: Player){
    self.playerOne = playerOne
    self.playerTwo = playerTwo
    //设置当前走棋的玩家
    self.curPlayer = self.playerOne
    }
    //棋局创建
    func compositionCreate(composition: Composition){
    self.composition = composition
    self.curChesses = composition.startChesses
    }
```

因为正在进行中的棋局只有一个，所以这里我们可以给棋局管理类添加一个单例，使用单例模式可以方便棋局数据的传递和管理，保证了棋局不依附于控制器，可以单独存在。在 CompositionManager 里加入单例的声明。

```
            //单例
    static let shared = CompositionManager()
```

这时我们可以在 HomeViewController 里选择模式的按钮被单击后加入相关代码，来完成单机双人对战模式时的初始数据创建。

```
          let object = self.cellArray[indexPath.row]
        let to_vc = CompositionViewController()
        //根据 key 值判断不同类型的游戏方式，后期做不同的处理
          switch object.key {
    case "outline":
          to_vc.type = 0
          let player1 = Player(id: "1", name: "玩家 1", avatar: "")
            let player2 = Player(id: "2", name: "玩家 2", avatar: "")
          CompositionManager.shared.playerJoin(playerOne: player1, playerTwo:
player2)
          let composition = Composition(id: "1", startChesses: CompositionUtil.
randomQue())
          CompositionManager.shared.compositionCreate(composition:
composition)
          break
        case "AI":

          break
        case "online":

          break
        default:
```

```
            break
        }
```

这里为了在控制器里区分当前的游戏模式，所以在 CompositionViewController 里声明了一个 type 属性代表当前的游戏模式。

```
    var type: Int = 0 //0:双人，1:人机，2:联网
```

3．CompositionManager 和视图之间的数据传递

为了实现 manager 和视图之间的数据传递，接下来需要定义一个协议，在协议中声明相关方法，这些方法就是处理了走棋相关的业务逻辑接口，它交给控制器去实现。

```
    protocol CompositionManagerDelegate: NSObjectProtocol {
            //游戏开始
            func gameBegin(_ manager: CompositionManager)
            //走棋
            func didMove(_ manager: CompositionManager, step: Step)
            //游戏结束
            func gameOver(_ manager: CompositionManager)
            //选择方向
            func showDirections(_ manager: CompositionManager, chess: Chess,
directions: [StepDirection])
            //改变玩家了
            func playerSwitched(_ manager: CompositionManager)
    }
```

在 CompositionManager 里声明代理属性：

```
    weak var delegate: CompositionManagerDelegate?
```

再定义一个游戏开始的方法。游戏开始时通过代理通知控制器，这样控制器监听到游戏开始，就可以进行页面棋子视图的加载了。

```
        //游戏开始
    func beginGame(){
      self.delegate?.gameBegin(self)
    }
```

还记得前面我们是如何将界面加上棋子的吗？我们是在 CompositionView 的 loadChess 方法里使用 for 循环加入了测试数据的。接下来我们就需要对它进行改写了，将棋局数据的生成交给 manager，这里只要接收棋局数据并加载相应的视图即可。

```
        //加载棋局
    func loadChess(_ chesses: [Chess?]){
      //先移除原来的棋子
      for btn in self.chessButtons{
        btn?.removeFromSuperview()
      }
      self.chessButtons.removeAll()
```

```
      for chess in chesses{
        if chess == nil{
          self.chessButtons.append(nil)
          continue
        }
        let chessBtn = ChessButton(chess!)
        chessBtn.addTarget(self, action: #selector(self.didChess(_:)), for: .
touchUpInside)
        self.chessButtons.append(chessBtn)
      }

      for (i, btn)in self.chessButtons.enumerated(){
        if btn == nil{
          continue
        }

        let x = CGFloat(i%4)*wh
        let y = CGFloat(i/4)*wh
        btn?.frame = CGRect(x: x+padding, y: y+padding, width: wh-padding*2,
height: wh-padding*2)
        self.chessboard.addSubview(btn!)
      }
    }
```

接下来我们要在控制器里实现 CompositionManagerDelegate 代理，并将棋局管理类单例的代理指向当前控制器。在 CompositionViewController 里完成以下代码。

```
    var compositionManager =  CompositionManager.shared

    func createGame(){
    self.compositionManager.delegate = self
    }
```

在 viewDidLoad()方法里加入以下代码。

```
    self.createGame()
```

使用 extension 扩展实现 CompositionManagerDelegate 代理。

```
    extension CompositionViewController: CompositionManagerDelegate{
            //游戏开始，初始化设置相关信息
            func gameBegin(_ manager: CompositionManager) {
            }

            func didMove(_ manager: CompositionManager, step: Step) {
            }
```

```
            func gameOver(_ manager: CompositionManager) {
            }

            func showDirections(_ manager: CompositionManager, chess: Chess,
directions: [StepDirection]) {
            }

            func playerSwitched(_ manager: CompositionManager){
            }
    }
```

当界面加载完成后开始游戏，我们可以重写 UIViewController 的 viewDidAppear 方法，这个方法就是在控制界面出现后调用的。

```
    override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)
    self.reload()
    }

    //开始游戏
    func reload(){
    self.compositionManager.beginGame()
    }
```

完成游戏开始后界面的修改，即实现相应的代理方法。当游戏开始时，将对应的游戏玩家信息载入玩家视图上，同时将开局数据加入棋局视图。

```
        func gameBegin(_ manager: CompositionManager) {
        self.playerOneView.player = manager.playerOne
        self.playerTwoView.player = manager.playerTwo
    self.compositionView.loadChess(manager.composition.startChesses)
        }
```

开局后就是界面的棋子等待玩家单击了，玩家单击了界面的棋子后，需要控制器监听到哪个棋子被单击了，并将被单击的棋子通过相应的方法传给 manager，然后在 manager 里处理相应的逻辑。

我们可以在 CompositionView.swift 文件定义一个协议来传递界面被单击的事件。

```
    protocol CompositionViewDelegate: NSObjectProtocol{
            func clickChess(_ view: CompositionView, chess: Chess)
            func clickDirection(_ view: CompositionView, chess: Chess,
direction: StepDirection)
    }
```

在 CompositionView 里声明代理属性。

```
    weak var delegate: CompositionViewDelegate?
```

在 CompositionViewController 控制器里声明 compositionView 时将相应的代理指向当前控制器。

```
    lazy var compositionView: CompositionView = {
    let v = CompositionView()
    v.delegate = self
    v.backgroundColor = UIColor.white
    return v
    }()
```

CompositionViewController 实现对应的协议和相关的代理方法。

```
    extension CompositionViewController: CompositionViewDelegate{
            //界面棋子被单击了，告诉 manager，处理相关逻辑
            func clickChess(_ view: CompositionView, chess: Chess) {
            }
            //棋子要走的方向按钮被单击了，告诉 manager，处理相关逻辑
            func clickDirection(_ view: CompositionView, chess: Chess,
direction: StepDirection) {
            }
    }
```

实现了这样的代理方法，就是为了将相关单击事件告诉 compositionManager，我们需要在 compositionManager 里定义相应的方法，这样就可以在代理方法里直接调用 manager 中的相应方法进行游戏的业务逻辑处理了。首先是有棋子被单击了。

```
        //棋子被单击了
    func clickedChess(_ chess: Chess){
      if chess.isShow{

      }else{
        self.move(chess, direction: .turn)
      }
    }
        //开始走棋逻辑
    func move(_ chess: Chess, direction: StepDirection){
      let step = Step(count: self.stepArr.count+1, playerId: self.curPlayer.id,
chess: chess, direction: direction)
     guard let curPosition = chess.position else {
        return
      }
      switch direction {
     case .turn:
        self.curChesses[curPosition.index]?.isShow = true
        break
      case .right:

        break
```

```
        case .left:

          break
        case .up:

          break
        case .down:

          break
        }
        self.stepArr.append(step)
        //走棋逻辑结束后，通知界面
        self.delegate?.didMove(self, step: step)
    }
```

下面我们完成翻棋的流程。当 manager 得知有棋子被单击时，首先判断被单击的棋子是否翻开了，如果没有翻开，则执行翻棋的代码。定义一个 move 方法，用于处理所有走棋的业务。根据当前棋子和走棋方向创建一个走棋记录，若是翻棋，则直接修改当前棋子数组里对应棋子的显示状态为 true，再将走棋记录加入记录数组 stepArr 中，最后调用协议方法，通知界面进行翻棋操作。

界面接收翻棋的通知，进行翻棋操作，在 CompositionView 中创建对应的方法。

```
    //走棋
    func move(_ step: Step, complish: @escaping ()->()){
        guard let curPosition = step.chess.position else {
          return
        }

        switch step.direction {
        case.turn:
          self.turn(curPosition, complish: complish)
          break
        case.right:
          break
        case.left:
          break
        case.up:
          break
        case.down:
          break
        }
    }
```

```
    //翻棋
func turn(_ position: CBPosition, complish: @escaping ()->()){
  self.chessButtons[position.index]?.chess?.isShow = true
  self.chessButtons[position.index]?.turnChess(complish)
}
```

在CompositionViewController中实现的协议代理方法didMove里调用界面走棋的方法。

```
func didMove(_ manager: CompositionManager, step: Step) {
//开始走棋动画
self.compositionView.move(step) {

}
}
```

4．走棋的业务逻辑实现

前面已经完成了翻棋的 manager 和界面间数据传递流程，接下来就是棋子已经翻开，如何实现走棋的业务逻辑了。当棋子已经翻开时，我们单击棋子后应是怎样的流程呢？棋子已经翻开，那么就需要判断被单击的棋子是否属于当前玩家的，如果不属于，则不能进行操作。我们之前已经在 CompositionManager 里声明了 curPlayer 属性，并在 playerJoin 方法里为其赋值为 playerOne。

轮流走棋的过程，其实也就是当前玩家不断替换的过程，因此可以在 Composition Manager 里定义一个玩家交替的方法。

```
    //改变当前玩家
func switchPlayer(){
  if self.curPlayer.id == self.playerOne.id{
    self.curPlayer = self.playerTwo
  }else{
    self.curPlayer = self.playerOne
  }
  self.delegate?.playerSwitched(self)
}
```

在界面执行完走棋动画后，自然就需要交换当前玩家了。可以在 Composition ViewController 中实现的协议方法 didMove 里的界面走棋动画方法的闭包函数内加入相关代码。

```
        self.compositionView.move(step) {
        self.compositionManager.switchPlayer()
}
```

当前玩家改变了，就需要在界面中体现出来，当前是轮到哪一个玩家走棋了，所以我们需要对玩家视图再进行相应处理。

我们需要在玩家视图 PlayerView 里再添加一个控件，可以用这个控件的显示和隐藏功能来区分当前走棋的玩家是谁。在 PlayerView 里声明一个属性。

```
 var curMoveLabel: UILabel!
```

接着在 createView 方法里对 curMoveLabel 实例化，并对相关属性进行设置。

```
self.curMoveLabel = UILabel()
self.curMoveLabel.textColor = Color.textL
self.curMoveLabel.font = Font.middle
self.curMoveLabel.text = "思考中..."
self.curMoveLabel.isHidden = true
self.addSubview(self.curMoveLabel)
```

分别在 snpLeft 和 snpRight 里对 curMoveLabel 添加不同的布局约束。

```
    func snpLeft(){
    self.curMoveLabel.snp.makeConstraints { (make) in
    make.left.equalTo(self.avatarImageView.snp.right).offset(10)
    make.bottom.equalTo(self.avatarImageView)
  }
    }

    func snpRight(){
  self.curMoveLabel.snp.makeConstraints { (make) in
    make.right.equalTo(self.avatarImageView.snp.left).offset(-10)
    make.bottom.equalTo(self.avatarImageView)
  }
}
```

如何判断当前被单击的棋子是否属于当前玩家。棋局开始时还不能确定当前玩家拥有哪种棋子，只有第一步完成后才能确定，那么我们就可以根据第一步来判断当前棋子是否属于当前的玩家。可以在 CompositionManager 里加入一个方法。

```
      //判断棋子是否是当前玩家可操作的
func chessIsCur(_ chess: Chess)->Bool{
  if self.stepArr.count == 0{
    return true
  }
  //获取第一步，第一步是决定玩家拥有哪一方棋子的
  let step = self.stepArr[0]
  if self.curPlayer.id == step.playerId{
    return chess.belong == step.chess.belong
  }else{
    return chess.belong != step.chess.belong
  }
}
```

然后给 PlayerView 声明一个当前是否在走棋中的属性，并在棋 didSet 方法里设置显示还是隐藏 curMoveLabel。

```
      var isMoving: Bool = false{
```

```
    didSet{
      self.curMoveLabel?.isHidden = !isMoving
    }
  }
```

最后在 CompositionViewController 的代理方法 playerSwitched 里具体实现玩家交换的逻辑。

```
    func playerSwitched(_ manager: CompositionManager){
    if manager.curPlayer.id == manager.playerOne.id{
      self.playerOneView.isMoving = true
      self.playerTwoView.isMoving = false
    }else{
      self.playerOneView.isMoving = false
      self.playerTwoView.isMoving = true
    }
  }
```

当然，我们在游戏开始时也是分配了玩家的，那么我们就在 gameBegin 的方法里也调用这个方法。

```
    func gameBegin(_ manager: CompositionManager) {
    self.playerOneView.player = manager.playerOne
    self.playerTwoView.player = manager.playerTwo

    self.compositionView.loadChess(manager.composition.startChesses)
    self.playerSwitched(manager)
  }
```

既然要走棋，就应先判断当前棋子有哪些方向可以走。如何判断当前棋子可以走哪些方向呢？其实就是对比目标位置的情况，如果目标位置不在棋盘内，或者目标位置有棋子，且棋子不能被当前棋子吃掉，那么就不能移动到目标位置，我们可以在 CompositionManager 里定义这样一个方法。

```
   func judgeCanMove(_ chess: Chess, newPosition: CBPosition)->Bool{
  //目标位置不在棋盘内返回 false
  if !newPosition.isInBoard{
        return false
  }
  //目标位置是否有棋子，没有返回 true，有则判断是否可以吃掉
  if let toChess = self.curChesses[newPosition.index]{
  return chess.canEat(toChess)
  }
  return true
  }
```

与之对应，我们需要判断当前棋子的上、下、左、右四个目标位置是否可以移动，并记录为一个可移动方向的数组。

```
        //当前棋子可以移动的所有方向
    func moveDirections(_ chess: Chess)->[StepDirection]{
      var directions = [StepDirection]()
        guard let curPosition = chess.position else {
          return directions
        }
        if !chess.isShow{
          return [.turn]
        }
        //右
        let rightPosition = curPosition>+
        if self.judgeCanMove(chess, newPosition: rightPosition){
          directions.append(.right)
        }
        //左
        let leftPosition = curPosition>-
        if self.judgeCanMove(chess, newPosition: leftPosition){
          directions.append(.left)
        }
        //上
        let topPosition = curPosition^+
        if self.judgeCanMove(chess, newPosition: topPosition){
          directions.append(.up)
        }
        //下
        let bottomPosition = curPosition^-
        if self.judgeCanMove(chess, newPosition: bottomPosition){
          directions.append(.down)
        }
        return directions
    }
```

已经翻开的棋子被单击后，就需要在 manager 里判断可以走的方向，然后告诉 view。不过判断方向前自然是要判断当前被单击的棋子是否属于当前的玩家。我们可以在 clickedChess 方法的 chess.isShow 里写下相应的逻辑代码。

```
          //不是当前玩家的棋子不可操作
    if !self.chessIsCur(chess){
                    return
    }
      //计算当前棋子有哪些可以走的方向
      let directions = self.moveDirections(chess)
      self.delegate?.showDirections(self,chess:chess,directions:directions)
```

计算了当前棋子可以走的方向后，通过代理方法传递给 CompositionViewController，在控制器里就可以控制界面展示相应的方向按钮了。接下来在 compositionView 里加入显示方向按钮的方法。

```
        //显示方向按钮
func showDirections(_ chess: Chess, directions: [StepDirection]){

  guard let curPosition = chess.position else{
    return
  }

  //右
  if directions.contains(.right){
    let rightPosition =  curPosition>+
    let x = CGFloat(rightPosition.x)*wh
    let y = CGFloat(rightPosition.y)*wh
    self.directionViews.directionRight.frame.origin = CGPoint(x: x, y: y)
    self.chessboard.addSubview(self.directionViews.directionRight)
  }else{
    self.directionViews.directionRight.removeFromSuperview()
  }

  //左
  if directions.contains(.left){
    let leftPosition =  curPosition>-
    let x = CGFloat(leftPosition.x)*wh
    let y = CGFloat(leftPosition.y)*wh
    self.directionViews.directionLeft.frame.origin = CGPoint(x: x, y: y)
self.chessboard.addSubview(self.directionViews.directionLeft)
  }else{
    self.directionViews.directionLeft.removeFromSuperview()
  }

  //上
  if directions.contains(.up){
    let topPosition =  curPosition^+
    let x = CGFloat(topPosition.x)*wh
    let y = CGFloat(topPosition.y)*wh
    self.directionViews.directionTop.frame.origin = CGPoint(x: x, y: y)
    self.chessboard.addSubview(self.directionViews.directionTop)
  }else{
    self.directionViews.directionTop.removeFromSuperview()
```

```
        }

        //下
        if directions.contains(.down){
          let bottomPosition =  curPosition^-
          let x = CGFloat(bottomPosition.x)*wh
          let y = CGFloat(bottomPosition.y)*wh
          self.directionViews.directionBottom.frame.origin = CGPoint(x: x, y: y)
          self.chessboard.addSubview(self.directionViews.directionBottom)
        }else{
          self.directionViews.directionBottom.removeFromSuperview()
        }

              //闭包监听走棋方向按钮被单击
        self.directionViews.didDirection { (direction) in
          self.delegate?.clickDirection(self, chess: chess, direction: direction)
        }

      }
```

通过闭包方法监听方向按钮被单击，然后调用当前的代理方法将单击事件通过控制器传递给 manager。在控制器 CompositionViewController 的 CompositionViewDelegate 代理方法里加入调用 manager 走棋逻辑的代码。

```
          //棋子要走的方向按钮被单击了，告诉 manager，处理相关逻辑
      func clickDirection(_ view: CompositionView, chess: Chess, direction:
StepDirection) {
        self.compositionManager.move(chess, direction: direction)
      }
```

接下来完善 manager 里的走棋逻辑。走棋逻辑也就是将当前位置的棋子移到新的位置，并且如果新的位置有棋子就将其移除，我们可以先写一个方法来完成棋子从当前位置替换到新位置的逻辑，在 CompositionManager 里加入下列代码。

```
          //走棋后棋子位置改变，改变当前棋盘的数组并返回被吃掉的棋子
      func curChessMove(curPosition: CBPosition, newPosition: CBPosition)->
Chess?{
        let beEat = self.curChesses[newPosition.index]
        self.curChesses[newPosition.index] = self.curChesses[curPosition.index]
        self.curChesses[newPosition.index]?.position = newPosition
        self.curChesses[curPosition.index] = nil
        return beEat
      }
```

这样，我们就可以在 CompositionManager 的 move 方法里走向不同方向时调用这个方法了。

```
        //开始走棋逻辑
    func move(_ chess: Chess, direction: StepDirection){
      var step = Step(count: self.stepArr.count+1, playerId: self.curPlayer.id,
chess: chess, direction: direction)
       guard let curPosition = chess.position else {
         return
      }

      var beEat: Chess?
      switch direction {
      case .turn:
        self.curChesses[curPosition.index]?.isShow = true
        break
        case .right:
      beEat = self.curChessMove(curPosition: curPosition, newPosition: curPosi
tion>+)
        break
    case .left:
      beEat = self.curChessMove(curPosition: curPosition, newPosition: curPosi
tion>-)
        break
        case .up:
      beEat = self.curChessMove(curPosition: curPosition, newPosition: curPosi
tion^+)
        break
        case .down:
      beEat = self.curChessMove(curPosition: curPosition, newPosition: curPosi
tion^-)
        break
      }
    step.beEat = beEat
      self.stepArr.append(step)
      //走棋逻辑结束后，通知界面
      self.delegate?.didMove(self, step: step)
    }
```

下面就是通知界面完成走棋的动画效果了。走棋的动画效果也就是一个 ChessButton 从当前位置移到一个新的位置，且新的位置若有 ChessButton 就将其移除，我们可以先在 CompositionView 里加入一个方法。

```
        func curChessMove(curPosition: CBPosition, newPosition: CBPosition,
complish: @escaping ()->()){
```

```
      guard let chessBtn = self.chessButtons[curPosition.index]  else {
        return
      }

      //走棋的动画
      UIView.animate(withDuration: 1.0, animations: {

        let x = CGFloat(newPosition.x)*self.wh
        let y = CGFloat(newPosition.y)*self.wh
        chessBtn.frame.origin = CGPoint(x: x+self.padding, y: y+self.padding)

        self.chessButtons[newPosition.index]?.removeFromSuperview()
      }) { (finish) in
        if finish{
          self.chessButtons[newPosition.index] = self.chessButtons[curPosition.
index]
          self.chessButtons[newPosition.index]?.chess.position = newPosition
          self.chessButtons[curPosition.index] = nil
          complish()
        }
      }
    }
```

最后自然也就是在 move 方法里完成不同方向时调用该方法的代码了。

```
        //走棋
    func move(_ step: Step, complish: @escaping ()->()){

      guard let curPosition = step.chess.position else {
        return
      }

      switch step.direction {
      case.turn:
        self.turn(curPosition, complish: complish)
        break
      case.right:
        self.curChessMove(curPosition: curPosition, newPosition: curPosition>+,
complish: complish)
        break
      case.left:
        self.curChessMove(curPosition: curPosition, newPosition: curPosition>-,
complish: complish)
        break
```

```
      case.up:
        self.curChessMove(curPosition: curPosition, newPosition: curPosition^+,
complish: complish)
        break
      case.down:
        self.curChessMove(curPosition: curPosition, newPosition: curPosition^-,
complish: complish)
        break
      }
    }
```

完成走棋功能后，需要告诉 manager，当前玩家已经完成了走棋功能，同时 manager 需要处理走棋后需要做的事情，之后就是要判断游戏的是否结束了。所以我们需要一个判断游戏是否结束的方法。什么才叫游戏结束呢？只要有一方的棋子被吃光，游戏肯定也就结束了，但可能还存在一种可能性，那就是有一方的棋子被堵死在角落里，没有棋子可以走了，也算是游戏结束。我们完成这样的一个方法，先用两个初始值为 0 的 Int 变量来记录当前两方可走棋子的数量，然后遍历当前的棋子数组，判断棋子是否存在或可以走棋。

```
        //走棋动画完成
    func chessMoved(){
      if self.judgeGameOver(){
        self.delegate?.gameOver(self)
      }else{
        self.switchPlayer()
      }
    }

        //判断游戏是否结束
    func judgeGameOver()->Bool{
      var count1:Int = 0
      var count2:Int = 0

      for chess in self.curChesses{

        //如果棋子不存在，则直接进入下个循环
        if chess == nil{
          continue
        }

        //如果棋子翻开了，但没有方向可走，则进入下个循环
        if chess!.isShow && self.moveDirections(chess!).count == 0{
          continue
        }
```

```
        //分别记录不同方的当前可走棋子数量情况
        if chess!.belong == 0{
          count1 += 1
        }else{
          count2 += 1
        }
      }
      //只要有一方当前可走棋子数量为 0，就代表游戏结束
      return count1 == 0 || count2 == 0
    }
```

我们需要在哪里去调用这个方法呢？我们可以在每次走棋结束后都先调用这个方法来判断游戏是否结束，若结束则弹出游戏结束的提示，若没有结束则交换玩家继续游戏。所以我们在 CompositionViewController 中 CompositionManagerDelegate 的代理方法 didMove 里对之前直接交换玩家的地方修改为告诉 manager 走棋结束，让其处理相关业务逻辑。

```
        func didMove(_ manager: CompositionManager, step: Step) {
      //开始走棋动画
      self.compositionView.move(step) {
        manager.chessMoved()
        }
    }
```

5．游戏结束的处理

当游戏结束后，我们需要弹出一个窗口，告诉用户结果，并可以让用户选择返回或重新开始游戏。manager 里判断游戏结束了，通过代理告诉 CompositionViewController，然后就是界面的事情了，这里我们使用系统自带的 UIAlertController 作为弹出窗口。

```
        func gameOver(_ manager: CompositionManager) {
      let curName = manager.curPlayer.name

      let alertView = UIAlertController(title: "游戏结束", message: "\(curName)
获胜啦", preferredStyle: .alert)
      let actionNext = UIAlertAction(title:"下一局",style:.default){(action)in
        self.newGame()
      }
      let actionBack = UIAlertAction(title:"返回",style:.default){(action)in
        self.back()
      }
      alertView.addAction(actionNext)
      alertView.addAction(actionBack)
      self.present(alertView, animated: true, completion: nil)
    }
```

添加游戏结束后的处理，要么返回到首页，要么开始新的一局游戏，开始新一局的游戏也就是重新生成随机的棋局，然后重写加载界面等。

```
    //返回
    func back(){
      self.dismiss(animated: true, completion: nil)
    }

    //新游戏
    func newGame(){
      let composition = Composition(id: "1", startChesses: CompositionUtil.
randomQue())
      CompositionManager.shared.compositionCreate(composition: composition)
      self.reload()
    }
```

6．setView 里按钮单击事件的添加

在控制器 CompositionViewController 的 createView 方法里给 setView 上的三个按钮添加单击事件。

```
        self.setView.backButton.addTarget(self, action: #selector(self.didBack
Button), for: .touchUpInside)
    self.setView.regretButton.addTarget(self, action: #selector(self.didRegret
Button), for: .touchUpInside)
    self.setView.surrenderButton.addTarget(self,  action:  #selector(self.did
SurrenderButton), for: .touchUpInside)
```

在 CompositionViewController 里定义相关的按钮单击方法。

```
    @objc func didBackButton(){
    }
    @objc func didRegretButton(){
    }
    @objc func didSurrenderButton(){
    }
```

实现单击返回按钮的功能，为了防止由于手误单击该按钮，在单击该按钮后先弹出对话框，让玩家再次确认。

```
          @objc func didBackButton(){
      let alertView = UIAlertController(title: "确认返回", message: "返回后游戏
就结束了", preferredStyle: .alert)
      let actionSure = UIAlertAction(title:"确定",style:.default){(action)in
        self.back()
      }
      let actionCancel = UIAlertAction(title:"取消",style:.cancel,handler:nil)
      alertView.addAction(actionSure)
```

```
        alertView.addAction(actionCancel)
        self.present(alertView, animated: true, completion: nil)
    }
```

实现单击认输按钮的功能，同样为了防止由于手误单击该按钮，在单击该按钮后先弹出对话框，让玩家再次确认，认输后让玩家选择重新开始、退出、取消。

```
        @objc func didSurrenderButton(){
        let alertView = UIAlertController(title: "提示", message: "确认认输",
preferredStyle: .alert)
        let actionNext = UIAlertAction(title:"重新游戏",style:.default){(action)in
          self.newGame()
        }
        let actionOut=UIAlertAction(title: "退出游戏",style:.default){(action)in
        self.back()
        }
        let actionCancel = UIAlertAction(title: "取消", style: .cancel, handler:
nil)
        alertView.addAction(actionNext)
        alertView.addAction(actionOut)
        alertView.addAction(actionCancel)
        self.present(alertView, animated: true, completion: nil)
    }
```

7．悔棋功能的添加

前面完成了整个走棋的过程，包括开局、翻棋、走棋、游戏结束的判断。下面完成走棋过程中的悔棋功能。首先分析悔棋功能该如何实现，悔棋也就是撤销当前棋局走的最后一步棋。我们在棋局管理类 CompositionManager 里有专门记录每步走棋的数据的数组 stepArr，可以根据数组的最后一个元素，先将这一步走的棋子还原，根据最后一个走棋记录，我们把当前的棋子数据更新，然后移除数组里的这条走棋记录，最后通知界面重新载入棋子即可。

首先在 CompositionManager 里加入一个悔棋的方法。

```
        //悔棋
    func regret(){
      //首先获取走棋记录最后一条数据
      guard let step = self.stepArr.last else{
        return
      }

      let curPosition = step.chess.position!
      self.curChesses[curPosition.index] = step.chess
      switch step.direction {
      case .turn:
```

```
        break
      case .right:
        self.curChesses[curPosition>+.index] = step.beEat
        break
      case .left:
        self.curChesses[curPosition>-.index] = step.beEat
        break
      case .up:
        self.curChesses[curPosition^+.index] = step.beEat
        break
      case .down:
        self.curChesses[curPosition^-.index] = step.beEat
        break
      }
      self.delegate?.didRegret(self)
      self.stepArr.remove(at: self.stepArr.count-1)
        //悔棋后也是要交换当前玩家的
        self.switchPlayer()
    }
```

给 CompositionManagerDelegate 协议添加一个新的方法，通知控制器悔棋了。

```
//执行了悔棋
 func didRegret(_ manager: CompositionManager)
```

接下来在控制器 CompositionViewController 里实现这个代理方法，调用 compositionView 的 loadChess 方法重新加载棋局棋子界面。

```
func didRegret(_ manager: CompositionManager){
self.compositionView.loadChess(manager.curChesses)
}
```

这里还需要做一个判断，就是如果上一步棋子是翻棋，应不允许悔棋，当前才刚开始游戏，也不能悔棋。我们可以在 CompositionManager 里在加一个判断是否可以悔棋的只读属性。

```
        //当前是否可以悔棋
    var canRegret: Bool{
      guard let step = self.stepArr.last else{
        return false
      }
      return step.direction != .turn
    }
```

在 CompositionViewController 的悔棋按钮单击方法里加入相应的代码。

```
        @objc func didRegretButton(){
    if self.compositionManager.canRegret{
      let alertView = UIAlertController(title: "提示", message: "确认要悔棋",
```

```
preferredStyle: .alert)
      let actionSure = UIAlertAction(title:"确定",style:.default){(action)in
        self.compositionManager.regret()
      }
      let actionCancel = UIAlertAction(title:"取消",style:.cancel,handler:nil)
      alertView.addAction(actionSure)
      alertView.addAction(actionCancel)
      self.present(alertView, animated: true, completion: nil)
    }else{
      let alertView = UIAlertController(title: "提示", message: "此时不支持悔棋
", preferredStyle: .alert)
      let actionSure = UIAlertAction(title: "确定", style: .cancel, handler: nil)
      alertView.addAction(actionSure)
      self.present(alertView, animated: true, completion: nil)
      }
    }
```

至此，单机双人对战功能就已经创建完成了。

24.6 人机对战的实现

人机大战总是那么令人着迷，吸引大量眼球，这也是人工智能的起源。而人机对战也是棋类游戏常见的功能。本案例以基础的 Alpha-Beta 算法来实现这样一个人机对战的功能，重点也就是展示这样一个功能的实现，并不对人工智能进行深入的研究。当然读者有兴趣的话，可以看完本案例后自己进行更深入的学习，来优化 AI 走棋的功能。

1．AI 算法初探

实现机器思考，也就是通过算法来模拟人类思考的过程。人类思考一般是在脑海里想着走了这步棋后对方会怎么走，然后自己又该怎么走，通过这样几层的思考，来判断出怎样走棋对自己最有利。

对于机器的算法，最直观的想法就是罗列出每一种可能的走棋方式，然后以树状图的形式不断迭代来模拟可能的走棋过程，进而判断对自己最有利的走法，这其实也就是一个树状搜索的过程。

那么机器又是如何在走了一步后来判断当前是对自己有利的情况呢？这就需要一个函数来评估当前的棋局情况，称之为局面评估函数。所谓局面评估函数是指根据当前棋面中存在的棋子情况，计算出一个值，来反映当前棋面的情况。

这样，理论上我们就可以把一步可能走的棋走后的局面评估函数值计算出来，然后比较这样一个值，得出最好的走法。但现实情况是思考深度的加深，数据量的庞大，会消耗很多事件进行计算。

那么计算机是如何进行搜索算法的呢？这里先引入一个概念，即“最小-最大”原理，这也是目前棋类游戏算法的基本思想。限于篇幅，这里就不再进行深入的阐述了。目前对弈类游戏算法都会有一个算法，称为 Alpha-Beta 剪枝算法。简单描述就是给一定的深度，循环递归调用走棋方法返回一个最好的走棋方式。下面直接看这样一个方法的伪代码。

```
            //思考算法（Alpha-Beta 剪枝算法）
      func think(alpha: Int, beta: Int, depth: Int)->Int{
            //深度为 0 了
        if depth == 0{
                    //返回局面评估函数的值
          returnevaluate()
        }
            //搜索当前所有的走棋方式
        let steps = self.searchAllSteps()
            //记录自己当前的局面值
            var myAlpha = alpha
            //遍历所有的走棋方式
        for step in steps{
          //走这个走法
                move(step)
          //走完这步棋获胜则直接返回，不需要再继续思考对方的走棋过程
          if gameOver{
            //撤销这步走棋（只是思考过程，所以走棋需要撤销）
            regret()
            //游戏结束，返回一个很大的值
            return 8888
          }

          //思考对手走棋(思考过程递归调用)，取反是为了获得自己的棋面值
          let otherVal = -self.think(alpha: -beta, beta: -myAlpha, depth:
depth-1)

          //撤销这步走棋（思考过对方的走棋情况再撤销）
          regret()
                if otherVal >= beta{
            //若得到大于最大值的情况，进行剪枝
            return nextMove
          }

    //若得到大于当前局面值的情况，进行交换
          if otherVal > myAlpha{
            myAlpha = otherVal
```

```
        }
      }
    return myAlpha
  }
```

这样一段代码只是构建了一个算法的基本数学模型，通过遍历当前的所有走法的返回值，来比较出一个较好的走法。当然这里也做了一些基本的处理，如走棋后游戏结束了，就说明当前玩家获胜，可以直接返回一个很大的值，不再继续搜索，减少计算的数量。同时若对方走棋后自己的局面值比当前最大值大，可以剪枝，不再继续计算，这样处理也是为了减少计算的次数。当然这里返回的也只是一个数字，并不能直接运用到走棋过程中去。接下来我们需要对其进行完善，包括其中的局面评估函数，获取所有可走棋的数据，以及一些其他的优化处理。

2．创建 AI 思考类

我们需要创建一个 AI 思考类，在这个类里处理和 AI 思考相关的一些算法。我们可以在 Composition 目录的 Manager 文件夹下新建一个 AIManager 文件.swift，如图 24-25 所示。

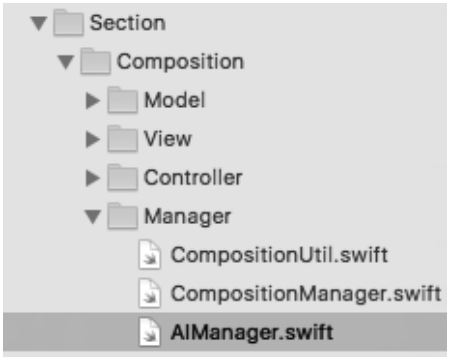


图 24-25

```
import UIKit
class AIManager: NSObject {
}
```

AI 思考过程肯定是需要知道当前的棋局情况的，然后进行模拟走棋搜索过程。我们可以在 AIManager 里加入一个 CompositionManager 属性。

```
class AIManager: NSObject {
       var curManager: CompositionManager!
}
```

3．局面评估函数

局面评估函数其实是棋类游戏中计算机思考最重要的一个指标，只有很准确地评估了当前的棋面情况，才能准确地获得最好的走棋方法。如果把一个对自己不利的局面评价成对自己有利的，那么再好的搜索算法也是枉然。写出一个很好的局面评估函数也是很困难的，这里我们也是简单地以局面棋子的大小来评估局面，对最小的棋子（鼠）的重要性稍

做了一点判断，这样的评估函数也只能说是初步评估了局面情况。我们首先在 AIManage 里写一个获得当前棋子价值的函数。

```
//当前棋局下棋子的价值，不同棋子的价值不同，不同情况下的相同棋子价值也不同
//定义价值如下：象 10、狮 8、虎 6、豹 5、狼 4、狗 3、猫 2
//若对方象还在：（我方象不在鼠 9）（我方象还在鼠 7）
//若对方象不在：鼠 1
func chessPrice(_ chess: Chess)->Int{
  switch chess.value {
  case 1:
    return 10
  case 2:
    return 8
  case 3:
return 6
  case 4:
    return 5
  case 5:
    return 4
  case 6:
    return 3
  case 7:
    return 2
  case 8://鼠
    //自己的象是否存在
    var myElephanExist = false
    //对方的象是否存在
    var otherElephanExist = false

    for chess in self.curManager.curChesses{
      if chess == nil{
        continue
      }
      if chess!.value != 1{
    continue
      }
      if self.curManager.chessIsCur(chess!){
        myElephanExist = true
      }else{
        otherElephanExist = true
      }
    }
    //如果对方的象存在
    if otherElephanExist{
      //我的象也存在返回 7，我的象不存在返回 9
```

```
        if myElephanExist{
          return 7
        }else{
          return 9
        }
      }else{
        return 1
      }
    default:
      return 0
    }
  }
```

接下来在 AIManage 里加入局面评估函数，也就是对当前局面所有棋子的值做一次计算。把属于自己的棋子值相加，不属于自己的棋子值相减。

```
  //局面评估函数
    func evaluate()->Int{
      var eValue = 0
      for chess in self.curManager.curChesses{
        if chess == nil{
          continue
        }
        if self.curManager.chessIsCur(chess!){
          eValue += self.chessPrice(chess!)
        }else{
          eValue -= self.chessPrice(chess!)
        }
  }
    return eValue
  }
```

4．搜索所有走法的函数

有了局面评估函数，接下来就是遍历所有的走法了，这样才能比较走棋后的局面情况。所以我们需要一个函数来返回当前局面下当前玩家的所有可能的走法。

```
      func searchAllSteps()->[Step]{
    var steps = [Step]()
    for chess in self.curManager.curChesses{
      //
      if chess == nil{
        continue
      }

      //棋子未翻开，记录棋子可翻
      if !chess!.isShow{
        let step = Step(count: self.curManager.stepArr.count+1, playerId:
```

```
self.curManager.curPlayer.id, chess: chess!, direction: .turn)
          steps.append(step)
          continue
        }

        //不是自己的棋子不可以走
        if !self.curManager.chessIsCur(chess!){
          continue
        }

        //获取当前棋子所有可以走的方向，循环加入走法记录里
        let directions = self.curManager.moveDirections(chess!)
        if directions.count == 0{
          continue
        }
        for direction in directions{
          self.curManager.move(chess!, direction: direction)
          let step = self.curManager.stepArr.last!
          steps.append(step)
          self.curManager.switchPlayer()
          self.curManager.regret()
        }
      }
      return steps
    }
```

5．走棋算法的实现

前面提到过通过递归计算出当前最佳走法的局面值方法，那么运用到实际的走棋过程中，我们需要获得的应是一个走法，而不是一个简单的数字。同时我们又需要记录每个走法后的局面值，可以定义一个结构的数据模型，在 Model 路径下新建一个 Move.swift 文件。

```
struct Move {
        var value: Int
        var step: Step
}
```

这样我们可以在 AIManager 里定义一个方法，来返回最佳的走法。

```
//思考算法
func think(alpha: Int, beta: Int, depth: Int, move: Move!)->Move{
}
```

其中 alpha 代表当前最小的期望值，beta 代表当前最大的期望值。depth 代表思考的深度，move 代表上一步思考的走法，最后返回最佳的走法。接下来我们就来完善这个方法。首先要搜索获得当前的所有走法，然后进行筛选，因为棋子没有翻开时是不知道棋子的值的，所以先把没有翻开的棋子筛选掉，然后在已经翻开的棋子里搜索走法。在 think 方法里加入相关代码。

```
            //搜索所有走法
            let steps = self.searchAllSteps()
            //过滤翻棋
            let moveSteps = steps.filter { (step) -> Bool in
        return step.direction != .turn
    }
            //记录最佳走法
    var niceStep: Step!
    var myAlpha = alpha
            //遍历所有走法
      for step in moveSteps{

      }
```

遍历走法里就是模拟走棋过程，需要根据 step 进行走棋逻辑，我们可以在 Composition Manager 里定义一个新的方法。

```
    //根据走棋数据走棋
    func move(_ step: Step){
    self.move(step.chess, direction: step.direction)
    }
```

下面完善 think 方法里遍历走棋过程的具体逻辑，首先执行走棋方法，然后判断走棋后游戏是否结束，若游戏结束了，则说明这个走法很好，那么就可以直接返回这个走法，并设置一个很大的值作为当前的局面值。若游戏没有结束，则先记录当前走法，在思考对手的可能的最好走法，然后比较对手走后自己的局面值。

```
          for step in moveSteps{
        //执行这个走法
        self.curManager.move(step)
        //走完这步获胜则直接返回，不需要再继续思考对方的走棋过程
        if self.curManager.judgeGameOver(){
          //撤销这步走棋（只是思考过程，所以走棋需要撤销）
          self.curManager.switchPlayer()
          self.curManager.regret()
          //游戏结束
          return Move(value: 8888, step: step)
        }
        self.curManager.switchPlayer()
        //走棋后记录当前的局面值
        let nextMove = Move(value: self.evaluate(), step: step)
        //思考对手走棋(思考过程递归调用)
        let otherMove = self.think(alpha: -beta, beta: -myAlpha, depth: depth-1,
move: nextMove)
    //取反是为了获得自己的局面值
        let otherVal = -otherMove.value
        //撤销这步走棋（思考过对方的走棋情况再撤销）
        self.curManager.regret()
```

```
        if otherVal >= beta{
          //若得到大于最大值的情况，则进行剪枝
          return nextMove
        }
        if otherVal > myAlpha{
          niceStep = step
          myAlpha = otherVal
        }
      }
```

若存在最佳走法，就返回这个走法，否则判断当前是否是思考的第一层。不是思考的第一层，就返回上一步走法和当前局面值，否则随机返回一个走法，我们可以给 AIManager 声明一个新的属性 treeDepth 来判断是否是第一层思考。

```
            if niceStep != nil{
        if myAlpha != alpha{
          return Move(value: myAlpha, step: niceStep)
    }
    }
            //如果不是根节点
        if self.treeDepth > depth{
          return Move(value: self.evaluate(), step: move.step)
        }
            //没有最佳根节点的走法，说明怎么走都得不到有利的局面，那么就随机返回一个走法（增加随机性）
        let ranInt = Int(arc4random() % UInt32(steps.count))
        return Move(value: myAlpha, step: steps[ranInt])
```

我们在 AIManager 里定义一个方法，通过这个方法传入当前的棋局情况，并通过闭包返回计算得到的最佳走法，关于这部分的整体代码如下。

```
    //思考深度
      var treeDepth = 5

      func think(_ manager: CompositionManager, complish: @escaping (_ move: Move)->()){
        self.curManager = manager
        let depth = self.treeDepth
        let curValue = -9999//一开始定义自己的局面值很低，这样就可以获得使自己损失最小的走法
        DispatchQueue.main.async {
          let move = self.think(alpha: curValue, beta: -curValue, depth: depth, move: nil)
          complish(move)
        }
      }
```

```
    //思考算法
    func think(alpha: Int, beta: Int, depth: Int, move: Move!)->Move{
      //深度为 0 就不再思考了，直接返回当前局面值
      if depth == 0{
        return Move(value: self.evaluate(), step: move.step)
      }
      //搜索所有走法
      let steps = self.searchAllSteps()
      //过滤翻棋
      let moveSteps = steps.filter { (step) -> Bool in
        return step.direction != .turn
      }

      //记录最佳走法
      var niceStep: Step!
      var myAlpha = alpha

      //遍历所有走法
      for step in moveSteps{
        //执行这个走法
        self.curManager.move(step)
        //走完这步获胜则直接返回，不需要再继续思考对方的走棋过程
        if self.curManager.judgeGameOver(){
          //撤销这步走棋（只是思考过程，所以走棋需要撤销）
          self.curManager.switchPlayer()
          self.curManager.regret()
          //游戏结束
          return Move(value: 8888, step: step)
        }
        self.curManager.switchPlayer()
    //走棋后记录当前的局面值
        let nextMove = Move(value: self.evaluate(), step: step)
        //思考对手走棋(思考过程递归调用)
        let otherMove = self.think(alpha: -beta, beta: -myAlpha, depth: depth-1,
move: nextMove)

        let otherVal = -otherMove.value

        //撤销这步走棋（思考过对方的走棋情况再撤销）
        self.curManager.regret()

        if otherVal >= beta{
```

```
            //截断
            return nextMove
          }
          if otherVal > myAlpha{
            niceStep = step
            myAlpha = otherVal
          }
        }

        if niceStep != nil{
          if myAlpha != alpha{
            return Move(value: myAlpha, step: niceStep)
          }
        }

        //如果不是根节点
        if self.treeDepth > depth{
          return Move(value: self.evaluate(), step: move.step)
        }

        //没有最佳根节点的走法，说明怎么走都得不到有利的局面，那么就随机返回一个走法（增加随机性，使得思考更加人性化）
        let ranInt = Int(arc4random() % UInt32(steps.count))
        return Move(value: myAlpha, step: steps[ranInt])
      }
```

有了这些简单的走棋思考算法，下面就将这些算法用到棋局中去。如何运用到棋局中去呢？首先我们应定义当前玩家中的一个是 AI 玩家，可以通过给 Player 的数据模型里声明一个新的属性 isAI 来代表是 AI 玩家。

```
struct Player {
        var id: String //玩家 id
        var name: String //玩家名称
        var avatar: String //玩家头像
        var isAI: Bool = false//代表 AI 玩家

        init(id: String, name: String, avatar: String) {
        self.id = id
        self.name = name
        self.avatar = avatar
        }
}
```

接下来我们只要在游戏数据载入时将原来的玩家 2 替换成 AI 玩家即可。我们在 HomeViewController 里单击按钮事件中 case"AI"的情况下加入新的代码。

```
        case "AI":
      to_vc.type = 1
      let player1 = Player(id: "1", name: "玩家 1", avatar: "")
      var player2 = Player(id: "2", name: "简单的计算机玩家", avatar: "")
      player2.isAI = true
      CompositionManager.shared.playerJoin(playerOne:player1,playerTwo:player2)

      let composition = Composition(id: "1", startChesses: CompositionUtil.
randomQue())
    CompositionManager.shared.compositionCreate(composition: composition)
      break
```

下面就是在玩家走棋结束、交换玩家后判断当前玩家是否是 AI 玩家，如果是，则执行相关的 AI 走法搜索过程。我们可以将 CompositionManager 里的 chessMoved 方法做相应的修改。

```
        func chessMoved(){
      if self.judgeGameOver(){
        self.delegate?.gameOver(self)
      }else{
        self.switchPlayer()
        if self.curPlayer.isAI{
          let manager = CompositionManager()
          manager.composition = self.composition
          manager.playerOne = self.playerOne
          manager.playerTwo = self.playerTwo
          manager.stepArr = self.stepArr
          manager.curChesses = self.curChesses
          manager.curPlayer = self.curPlayer

          let ai = AIManager()
          ai.think(manager, complish: { (move) in
            self.move(move.step)
          })
        }
      }
    }
```

最后我们需要控制界面，当 AI 玩家走棋时当前玩家不能再操作棋局。那么我们就可以在玩家交换了后的 CompositionManagerDelegate 协议代理方法 playerSwitched 添加一层判断，判断当前玩家是否是 AI 玩家，若是 AI 玩家就禁止玩家单击棋局视图。

```
        func playerSwitched(_ manager: CompositionManager){
      if manager.curPlayer.id == manager.playerOne.id{
        self.playerOneView.isMoving = true
        self.playerTwoView.isMoving = false
```

```
    }else{
      self.playerOneView.isMoving = false
      self.playerTwoView.isMoving = true
    }

    if manager.curPlayer.isAI{
      self.compositionView.isUserInteractionEnabled = false
    }else{
      self.compositionView.isUserInteractionEnabled = true
    }

  }
```

人机对战部分到此创建完毕，这里我们也只是以简单的方式处理了 AI 部分，其实还有很多地方可以优化。但本案例的目的还是想让读者熟悉 Swift 4 在项目中的应用，所以对更深奥的内容不过多研究。

24.7　联网对战的实现

对战类游戏，联网对战也是必不可少的一个模式。既然是联网对战，那么自然少不了网络请求的处理，而且联网对战也就是涉及发送消息和接收消息，只有 http 请求自然是不够的，这里我们使用了 WebSocket 来搭建后台服务器。当然使用其他方式也可以，socket 或 xmpp 协议等。

学习本节内容，还需要有后台的配合，如果读者自己能搭建后台服务器则最好，如果不能则只用一些测试数据来模拟收、发消息的过程也是可以的。我们实现服务器和客户端通信的整个过程，如图 24-26 所示。

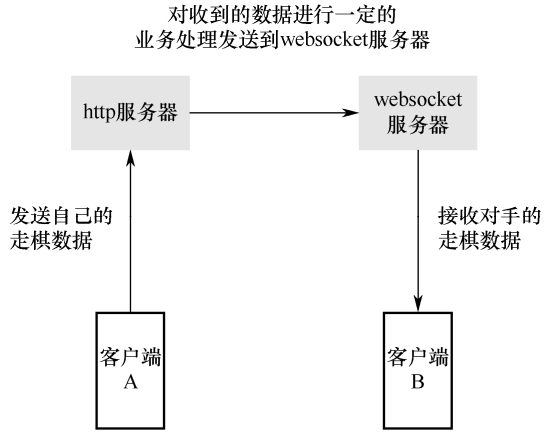


图 24-26

后台采用业务服务器+websocket 转发服务器的方式，本案例对于后台的具体数据业务处理不做详细叙述，重点在于数据的传输和处理、http 请求发送数据、websocket 接收数据、数据发送和接收时如何处理等。

1．http 请求的封装

和后台交换最常见的请求协议也就是 http 请求了，系统已为我们提供了比较全面的请求方法，但在实际项目中运用起来可能针对性不强。所以在实际开发时，很多项目可能会使用第三方封装好的库，或者对系统提供的方法自己简单地进行一次性封装。由于本案例中只是用到了比较常见和简单的请求方式，以及常见的 JSON 格式返回参数，所以我们就带领读者简单地封装一个网络请求的工具类。

这里使用系统为我们提供的网络请求类 URLSession，当然这不是唯一的方式，早在 URLSession 出来前还有 URLConnection 等网络处理类，为何选择 URLSession 呢？简单地说，就是因为觉得它比较强大、好用，这里我们仅用到其最基本的用法，建立一个请求数据的任务。

```
        let task = URLSession.shared.dataTask(with: URLRequest) { (data,
response, error)in


    }
```

这个 dataTask 有一个 URLRequest 参数，然后通过闭包返回数据结果。我们先看下如何声明 URLRequest 参数，一般的请求为接口地址、请求方式、传入参数。URLRequest 其实就是用来设置这些参数的。我们可以根据这三点定义一个方法，该方法传入这三个参数。我们先创建一个请求工具类 NetworkUtil，这个类放在我们前面创建的 Common 的 Util 路径下，如图 24-27 所示。

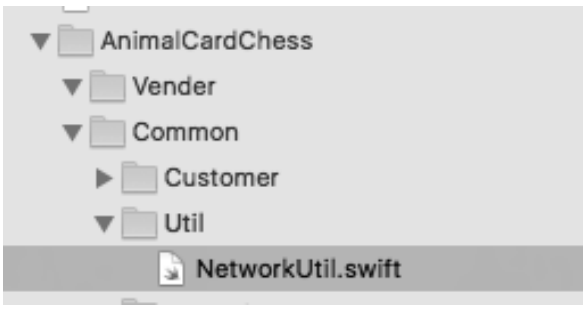


图 24-27

定义一个请求方法 request 并传入三个参数，代码如下。

```
    class NetworkUtil: NSObject {
            func request(url: URL, method: String, parameter: [String: Any]){


            }
    }
```

这里 method 用的是 String 类型，但其实 method 可能就是那么固定的几种，而且这里

只会用到最基本的 GET 和 POST 方式，所以我们可以定义一个枚举类型来代表 method。

```
enum NetMethod: String{
        case GET = "GET"
        case POST = "POST"
}
```

接下来就是 URL。在正常情况下，我们拿到的 API 接口其实就是一个字符串，而且如果是在 get 请求的情况下，可能还要将参数拼接在请求地址后面，所以我们可以直接传入字符串，在方法里再转成 URL，最后参数使用了[String: Any]这样一个字典类型，为了更方便地表示我们可以用 typealias 关键字定义一个类型。

```
typealias Parameter = [String: Any]
```

另外，我们可以给 request 方法的参数加上一些常用的默认值，这样可以省去一些不必要的代码。

```
 func request(urlStr: String, method: NetMethod = .POST, parameter:
Parameter? = nil){


}
```

我们时 dataTask 的网络处理方法是闭包返回了请求的结果的，而且结果包括三部分：(Data?, URLResponse?, Error?)，对此我们可以定义一个类来对结果进行简单的封装。

首先可以把 Data 结果直接返回我们想要的数据类型，JSON 是比较常用的后台返回数据类型，这里我们和后台规定好返回数据一定是 JSON 对象格式，对应的也就是字典类型。

而且 http 请求返回的结果也都是 URLResponse 的子类 HTTPURLResponse，这个子类就包括了请求结果的状态码 statusCode。

接着是 Error 的处理。Swift 的 Error 是一个空协议，我们可以简单地定义一个枚举来实现这个协议，并设置一些常见的错误类型。这样我们就可以定义一个类来处理返回结果了。

我们可以在 request 里添加一个闭包返回请求后的结果参数，最后我们只要对请求的参数和方式做一定的处理，就能完成一个简单的网络数据请求的封装了。这里由于是作为一个工具类来使用的，所以将一些相关的类都放在了一个文件里。为了避免不必要的误解，下面就列出这个文件里的所有代码。

```
  import UIKit

  enum NetMethod: String{
    case GET = "GET"
    case POST = "POST"
}

  typealias Parameter = [String: Any]

  class NetworkUtil: NSObject {
        //单例
```

```
        static let shared = NetworkUtil()
        var timeoutInterval: TimeInterval = 15.0

        func request(urlStr: String, method: NetMethod = .POST,  parameter:
Parameter? = nil, handler:@escaping (_ response: NetResponse)->()){

          var urlString = urlStr
          //如果是 get 方式，且有参数，则进行字符串拼接
          if method == .GET && parameter != nil{
            //调用字符串拼接的方法
            if let parStr = self.parameterStr(parameter!)
              .addingPercentEncoding(withAllowedCharacters:CharacterSet.urlQu
eryAllowed){
              urlString = "\(urlStr)?\(parStr)"
            }
          }
          guard let url = URL(string: urlString) else {
            //url 格式错误
            handler(NetResponse(error: .url))
            return
          }

          var request: URLRequest = URLRequest(url: url)
          request.httpMethod = method.rawValue
          request.timeoutInterval = timeoutInterval
          //如果是 POST 方式，则设置 httpBody 的值
          if method == .POST{
            let data = self.parameterStr(parameter!).data(using: String.Encod
ing.utf8)
            request.httpBody = data
          }
          let task = URLSession.shared.dataTask(with: request) { (data, response,
error) in
            DispatchQueue.main.async {
              //请求结果
              handler(NetResponse(response: response, data: data, error: error))
            }
          }
          task.resume()
        }

        //拼接请求参数为字符串
        func parameterStr(_ parameter: Parameter)->String{
```

```
        var str = ""
        for key in parameter.keys{
          var value = parameter[key]
          if value == nil{
            value = ""
          }
          str += "\(key)=\(value!)&"
        }
        return str
    }
}

enum NetError: Error{
    case url //请求地址格式出错
    case netWork //网络异常
    case noSource //无法访问资源
    case serverWrong //服务器出错
    case parameter //参数格式错误
    case other //其他错误
}

class NetResponse: NSObject{

    var response: HTTPURLResponse?
    var error: NetError?
    var parameter: Parameter?

    init(error: NetError) {
        self.error = error
    }

    init(response: URLResponse?, data: Data?, error: Error?){
        super.init()
        if error != nil{
            self.error = .netWork
            return
        }
        guard let httpRespose = response as? HTTPURLResponse else{
            self.error = .serverWrong
            return
        }
        self.response = httpRespose
```

```
        if httpRespose.statusCode == 404{
          self.error = .noSource
          return
        }
        if httpRespose.statusCode == 500{
          self.error = .serverWrong
          return
        }
        //返回参数不等于 200 设置为其他错误
        if httpRespose.statusCode != 200{
          self.error = .other
          return
        }
        //不存在 data
        if data  == nil{
          self.error = .serverWrong
          return
        }
        //data 解析成字典
        if let dict = (try? JSONSerialization.jsonObject(with: data!, options:
JSONSerialization.ReadingOptions.mutableContainers)) as? Parameter{
          self.parameter = dict
        }else{
          self.error = .parameter
        }
      }

    }
```

由于本案例也只是使用到了简单的网络数据请求，所以对于文件的上传和下载、或者其他格式的请求就不做太多的封装了，这里提供的就是一个简单而又基本的数据请求工具类。读者若有兴趣，也可以做更全面的网络请求封装。

网络请求封装完成后，若想要 App 能够进行网络请求，还需要在 Info.plist 文件里进一步配置，在文件里加入 App Transport Security Settings 的 key 值，并在其下增加 Boolean 类型的 Allow Arbitrary Loads 字段，设置其值为 YES。

2．登录功能的实现

既然是联网对战，也就需要服务器记录当前玩家用户，然后根据用户的 ID 来保证走棋数据传输的准确性。所以我们需要实现用户登录的功能，当然实际项目中可能还有用户注册等功能。这里为了简化操作，后台只提供一个登录接口。在实际项目中，后台自然就需要提供一个相对的接口文档，但对于后台的具体实现，本案例就不做过多讲解了。首先我们可以看下登录接口的文档，如图 24-28 所示。

- 用户登录接口
- `http://xxx.com/api/user/login`
- POST

请求参数说明

参数名	必选	类型	说明
account	是	string	账号
password	是	string	密码

返回参数说明

参数名	类型	说明
flag	int	请求结果标志：1成功，0失败
message	string	返回结果
user	Object	用户对象
id	string	用户id
nickname	string	昵称
avatar	string	头像地址

返回示例

```
{
  "flag": 1,
   "message": "请求成功",
  "user": {
    "id": "1",
    "nickname": "斗兽菜菜",
    "avatar": "http://xxx.com/upload/avatar/xxxx.png",
  }
}
```

图 24-28

从文档中可以看出，我们请求接口的地址，接口方式 POST，已经请求的参数是 account 和 password，请求返回的 JSON 格式数据，flag 代表成功或失败，message 则是成功或失败的说明，最后是 user 对象的具体数据。

知道了这些内容，下面就需要一个登录界面了，还有管理当前登录用户的一个类。我们先在 Section 下新建一个 User 模块，并创建相关文件夹和文件，如图 24-29 所示。

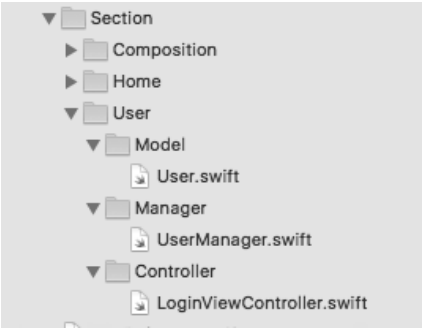


图 24-29

首先完成登录界面搭建。登录界面有三个控件：账号输入框、密码输入框、登录按钮。让 LoginViewController 继承 BaseViewController，然后完成界面的搭建，效果如图 24-30 所示。



图 24-30

```
      import UIKit

class LoginViewController: BaseViewController {

  //账号输入框
  lazy var accountTextField: UITextField = {
    let tf = UITextField()
    tf.backgroundColor = UIColor.white
    tf.textAlignment = .center
    tf.textColor = Color.textH
    tf.font = Font.middle
    tf.placeholder = "请输入账号"
    return tf
  }()

  //密码输入框
  lazy var pwdTextField: UITextField  = {
```

```
        let tf = UITextField()
        tf.backgroundColor = UIColor.white
        tf.textAlignment = .center
        tf.textColor = Color.textH
        tf.font = Font.middle
        tf.placeholder = "请输入密码"
        return tf
      }()

      //登录按钮
      lazy var loginButton: UIButton = {
        let btn = UIButton()
        btn.setTitle("登录", for: .normal)
      btn.setTitleColor(Color.textH, for: .normal)
        btn.backgroundColor = UIColor.white
        btn.addTarget(self, action: #selector(self.didLoginButton), for: .touch
UpInside)
        return btn
      }()

      override func viewDidLoad() {
        super.viewDidLoad()
        self.createView()
        self.createSnp()
      }

      override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
      }

      func createView(){
        self.view.addSubview(self.accountTextField)
        self.view.addSubview(self.pwdTextField)
        self.view.addSubview(self.loginButton)
      }

      func createSnp(){
        self.accountTextField.snp.makeConstraints { (make) in
          make.top.equalToSuperview().inset(80)
          make.centerX.equalToSuperview()
          make.width.equalToSuperview().multipliedBy(0.75)
          make.height.equalTo(44)
        }
        self.pwdTextField.snp.makeConstraints { (make) in
          make.top.equalTo(self.accountTextField.snp.bottom).offset(10)
```

```
        make.centerX.equalToSuperview()
        make.width.equalTo(self.accountTextField)
        make.height.equalTo(44)
      }
      self.loginButton.snp.makeConstraints { (make) in
        make.top.equalTo(self.pwdTextField.snp.bottom).offset(20)
        make.centerX.equalToSuperview()
        make.width.equalTo(self.accountTextField)
        make.height.equalTo(44)
      }
    }

    @objc func didLoginButton(){

    }

}
```

完成了界面的搭建后，就是具体登录功能的实现了。我们先定义当前用户的数据模型 User。

```
struct User: Codable {
        var id: String
        var nickname: String
        var avatar: String
}
```

这里我们让 User 遵循 Codable 协议，这个协议是 Swift 4 的新特性，遵循这个协议我们可以更方便地将后台返回的 JSON 数据进行序列化和反序列化。接下来我们就来看下具体如何使用。我们首先完成 UserManager 的创建。

```
  import UIKit
  class UserManager: NSObject {
        //单例
        static let shared = UserManager()
        //当前用户
        var user: User?

        //登录方法
        func login(_ account: String, password: String){
        }
}
```

使用我们之前封装好的网络请求工具，实现具体的登录功能。在此之前，我们还需要新建一个。swift 文件用来存放所有的接口地址，以方便对接口的统一管理。我们在 General 的 Marco 目录下新建一个 APIDefine.swift 文件，并加入登录接口地址。

```
import Foundation
```

```
    struct APIDefine{

            //接口地址的根路径
            static let BasePath = "http://192.168.2.172:8888/appapi"

            struct User {
            //登录接口
            static let login = "\(BasePath)/user/login"
            }
    }
```

我们同样使用 static let 来定义常量。这里的 BasePath 是后台接口的根地址。同时我们可以通过 struct 的嵌套对接口地址进行分组，将登录接口分在 User 组下。接下来就可以在 login 方法里进行登录接口的调用了。

```
        func login(_ account: String, password: String){

      let parameter: Parameter = ["account": account,
                     "password": password]

      NetworkUtil.shared.request(urlStr:  APIDefine.User.login,  parameter:
parameter) { (response) in

      }
    }
```

调用接口后就是返回数据的处理了，如何处理返回数据呢？首先我们可以给 login 方法加上一个闭包返回参数，然后对网络请求结果的 response 进行相应数据处理和判断，以判断登录是否成功，登录成功后将当前登录用到信息存储到 UserManager 的 user 属性中。我们可以试图先获取 response.parameter 里的 user 字段数据。

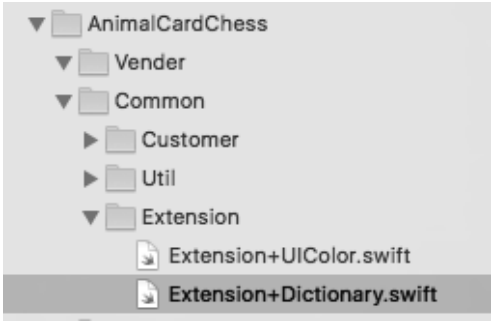


图 24-31

```
        if let dict = response.parameter?["user"] as? Parameter{

    }
```

若存在对应的 user 字段数据，那么我们就可以生成 User 对象了。我们要将字典数据通

过 Codable 协议的实例化方法转换成 User 对象，就先要将字典类型转换成 JSON 字符串的 Data。我们可以给 Dictionary 加一个扩展方法。在 Common 的 Extension 目录下新建一个 Extension+Dictionary.swift 文件。

```
    extension Dictionary{
            func data()->Data?{
            let data = try? JSONSerialization.data(withJSONObject: self,
options: JSONSerialization.WritingOptions.prettyPrinted)
            return data
             }
    }
```

接着完成 UserManager 里的 login 方法。

```
         func login(_ account: String, password: String, completion: @escaping
(_ errorMsg: String?)->()){

        let parameter: Parameter = ["account": account,
                        "password": password]

      NetworkUtil.shared.request(urlStr: APIDefine.User.login, parameter:
parameter) { (response) in

        if let dict = response.parameter?["user"] as? Parameter{
          let data = dict.data()!
          do{
            self.user = try JSONDecoder().decode(User.self, from: data)
            completion(nil)
          }catch{
            completion("登录失败")
          }
        }else{
          completion("登录失败")
        }
      }
    }
```

完成了这些准备工作，接下来就是 LoginViewController 里的单击按钮功能的具体实现了。首先我们是从 HomeViewController 控制器跳转到 LoginViewController 控制器的。什么情况下跳转呢？应在单击联网对战按钮时，先判断当前用户是否登录，若没有登录就跳转到登录界面。我们在 HomeViewController 里的单击事件方法中通过能否获取到当前用户来判断是否登录了。

```
        case "online":
    if let user = UserManager.shared.user{

    }else{
      let to_vc = LoginViewController()
```

```
    self.present(to_vc, animated: true, completion: nil)
  }
  return
```

最后我们完善 LoginViewController 里的登录按钮单击事件。

```
      @objc func didLoginButton(){
    guard let account = self.accountTextField.text, account != "" else{
      print("账号没有填写")
      return
    }
    guard let password = self.pwdTextField.text, password != "" else{
      print("密码没有填写")
      return
    }
    UserManager.shared.login(account, password: password) { (errorMsg) in
      if errorMsg == nil{
        //登录成功
        self.dismiss(animated: true, completion: nil)
      }else{
        //登录失败
        print(errorMsg!)
      }
    }
  }
```

3．WebSocket 第三方库的接入和使用

至此，完成了登录功能，使用的是 http 协议的网络请求方式。下面，我们需要对接 WebSocket，实现服务器主动发送数据给客户端的功能。这里我们也是使用已经有人封装好的第三方 WebSocket 库 Starscream，在 github 的地址是 https://github.com/daltoniam/Starscream。在 Podfile 文件里加入 pod 'Starscream'，然后通过终端 pod install，完成 WebSocket 第三方框架 Starscream 的导入。

```
platform :ios, '9.0'
use_frameworks!

target 'AnimalCardChess' do
       pod 'SnapKit', '~> 4.0.0'
       pod 'Starscream'
end
```

当然，这里也是需要后台配合的，本案例服务器端语音使用的是 PHP，WebSocket 也是使用第三方开源框架 GatewayWorker，只要读者有一定的 PHP 基础，搭建起来也是很简单的。服务器端搭建好后会给客户端一个与服务器端建立起连接的 WebSocket 地址，这里我们使用的局域网地址是 ws://192.168.2.172:8384/。

接下来我们看看是如何通过 Starscream 这个框架和服务器建立连接的。这个框架使用起来也是非常简单的。我们可以新建一个类来管理 WebSocket 和服务器端的通信。在 User

模块的 Manager 目录下新建一个 MessageManager.swift 文件。

```
    import UIKit
    import Starscream

    class MessageManager: NSObject{
            static let shared = MessageManager()
            var socket: WebSocket!

            override init() {
            super.init()
            self.socket = WebSocket(url: URL(string: "ws://192.168.2.172:
8384/")!)
            self.socket.delegate = self
            }
    }

    extension MessageManager: WebSocketDelegate{
            func websocketDidConnect(socket: WebSocketClient) {
            print("连接成功")
            }

            func websocketDidDisconnect(socket:WebSocketClient,error:Error?){
            print("断开连接")
            }

            func websocketDidReceiveMessage(socket: WebSocketClient, text:
String) {
            print("收到消息: ", text)
            }
            func websocketDidReceiveData(socket: WebSocketClient, data: Data){

            }
    }
```

我们只要首先导入 import Starscream 框架，然后在 MessageManager 里声明 WebSocket 属性 socket。在这个类实例化时给 WebSocket 加入通过 url 地址进行实例化，并添加代理，然后 MessageManager 实现 WebSocketDelegate 协议里的方法。WebSocketDelegate 协议里只有四个相关的方法，分别是 websocketDidConnect 连接成功、websocketDidDisconnect 连接断开、websocketDidReceiveMessage 收到字符串消息、websocketDidReceiveData 收到 Data 消息。

和 http 请求返回参数一样，后台给我们的数据也是 JSON 格式。连接成功后，后台会为当前客户端创建一个临时的 cliendId，并将这个 cliendId 发送给客户端。然后客户端需要将这个 cliendId 和当前用户 id 发送给业务服务器进行绑定。之后业务服务器端根据当前用户 id 就可以将消息发送到对应的客户端去了。

我们先来看看连接成功后，WebSocket 服务器发送过来的 cliendId 消息格式是怎样的。

```
 {"type":"init",
  "client_id":"7f0000010b5500000001"}
```

对 WebSocket 发来的数据，客户端进行处理。

```
    func websocketDidReceiveMessage(socket:WebSocketClient,text:String){
  print("收到消息: ", text)
  let data = text.data(using: String.Encoding.utf8)
  if let dict = (try? JSONSerialization.jsonObject(with: data!, options:
JSONSerialization.ReadingOptions.mutableContainers)) as? [String: Any]{
    print("reslut: ", dict)
    self.handleResponse(dict)
  }
}
```

此时服务器发来的 JSON 数据，type 代表数据类型。client_id 代表当前客户端分配的 id。接下来就是通过后台 http 接口进行客户端 id 和用户 id 的绑定。先在 APIDefine 里的 User 下加入绑定的接口。

```
      struct User {
    //登录接口
    static let login = "\(BasePath)/user/login"

    //绑定客户端 id
    /*
    * 客户端 id：client_id
    * 用户 id：uid
    */
    static let bindclient = "\(BasePath)/user/bindclient"
}
```

在 MessageManager 里加入处理方法，以及绑定客户端 id 的方法。

```
      //处理消息数据
func handleResponse(_ result: [String: Any]){
    guard let type = result["type"] as? String else{
        return
    }

    if type == "init"{
        self.client_id = result["client_id"] as? String
        self.bindClient()
    }
}

func bindClient(){
    if UserManager.shared.user == nil {
        return
    }
```

```
        if self.client_id == nil {
            self.connect()
            return
        }

        let parameter: Parameter = [
            "client_id"   : self.client_id!,
             "uid"   : self.client_id!,
            ]

        NetworkUtil.shared.request(urlStr: APIDefine.User.bindclient, parameter:
parameter) { (response) in
            guard let flag = response.parameter?["flag"] as? Int else{
                print("绑定失败")
                return
            }
            if flag == 1{
                print("绑定成功")
            }else{
                print("绑定失败")
            }
        }
    }
```

那么我们什么时候开始连接 socket 服务器和绑定客户端 id 呢？我们可以在用户登录成功后，进行这些操作，并在绑定成功后才算用户登录成功。我们需要在 UserManager 类里加入相关代码。

```
            func connectSocket(){
    MessageManager.shared.connect()
            }
```

4．匹配对战功能的实现

登录成功，也就连接了服务器，接下来，就可以进行联网对战了，联网对战第一步还是要匹配到对战的用户才可以，后台需要给我们提供一个进行匹配的接口。

这个接口会返回匹配的结果，其中 type 字段如果是 wait，则代表等待其他玩家加入，此时需要监听 websocket，当收到 websocket 匹配成功的消息时，把开局的相关数据（棋局+对手玩家）发送过来。然后生成棋局，开始对战。type 字段如果是 start，则会把对应的开局数据一起返回，就可以直接开局了。具体返回的开局数据事例如图 24-32 所示。

其中 player_one 和 player_two 是两个玩家的信息，其中一个是自己，start_position 是长度为 16 的 Int 数组，数据为 1～8、11～18 随机排列的数据，我们按照其在数组中的顺序代表棋子在棋盘中的位置，根据这些数据就可以生成棋局了。



```
["result": {
  id = 33;
  "player_one" =     {
      avatar = "http://xxx.com/upload/avatar/xxxx.png";
      id = 101;
      nickname = "斗兽菜菜";
  };
  "player_two" =     {
      avatar = "http://xxx.com/upload/avatar/xxxx.png";
      id = 102;
      nickname = "斗兽牛牛";
  };
  "start_position" =     (
      15,4,12,17,16,18,1,5,2,6,7,13,3,14,8,11
  );
  type = start;
}, "flag": 1, "msg": ]
```

- 匹
- ht
- PC

请求参

参数

uid

返回参

参数名	类型	说明
flag	int	请求结果标志：1成功，0失败
message	string	返回结果
result	Map	JSON对象
type	string	类型，wait：等待匹配，start：匹配成功
id	string	棋局id
player_one	对象	玩家1
player_two	对象	玩家2
start_position	数组	开局的棋局数组

图 24-32

首页单击联网对战按钮后，我们需要对其做相关的处理，加上开始匹配的方法。

```
        case "online":
      if UserManager.shared.loginStatus == .logined{
        self.startMatch()
      }else if UserManager.shared.loginStatus == .logining{
        print("连接中，请稍后。")
}else{
        let to_vc = LoginViewController()
        self.present(to_vc, animated: true, completion: nil)
      }
      return
```

之后我们在 APIDefine 文件里加入后台 API 接口地址。

```
struct APIDefine{

        //接口地址的根路径
        static let BasePath = "http://192.168.2.172:8888/appapi"
```

```
        struct Composition {
    //开始匹配
    /*
    * 用户 id：uid
    */
    static let matching = "\(BasePath)/composition/matching"
        }
    }
```

完善开始匹配的方法，也就是调用匹配接口，然后处理返回的数据。在 HomeView Controller 里定义新的方法 startMatch。

```
      //开始匹配
    func startMatch(){
        guard let user = UserManager.shared.user else{
            return
        }

        let parameter: Parameter = ["uid": user.id]

        NetworkUtil.shared.request(urlStr:    APIDefine.Composition.matching,
parameter: parameter) { (response) in
            guard let result = response.parameter?["result"] as? Parameter else{
                return
            }

    guard let type = result["type"] as? String else{
                return
            }

            if type == "wait"{
                //匹配中，开始检测匹配结果
               return
            }

            if type == "start"{
                //匹配成功，开局
               return
            }
        }
    }
```

这个方法里我们处理了返回结果，对 type 字段进行判断。我们可以先完成匹配成功的情况，匹配成功，要将后台返回的棋局相关数据进行处理，然后跳转至棋局控制器，进行游戏。考虑我们之前是在 CompositionUtil 类里写了随机生成棋局中棋子的布局方法，但联网对战棋局中棋子的布局是根据后台返回数据生成的，那么我们可以在 CompositionUtil 里加一个方法，来处理后台返回数据生成棋局。

```
        //根据 Int 数组生成棋局
    class func createQue(_ bvs: [Int])->[Chess]{
        let positionArr = bvs.enumerated().map({ (bv, i) -> Chess in
            var chess = Chess(belong: bv/10, value: bv%10)
            chess.position = CBPosition(index: i)
            return chess
        })
        return positionArr
    }
```

接着在 HomeViewController 里定义一个 startOnlineGame 方法，在一些解析数据的地方尽量进行判断，以防止后台返回数据出错，导致程序崩溃，我们拿到数据后，还要判断哪一个玩家数据是自己，将其置于棋局玩家 1 中，以保证游戏界面棋盘的下方是自己，对方在游戏棋盘的上方。

```
        //根据开局数据开局
    func startOnlineGame(_ dict: Parameter){
        guard let user = UserManager.shared.user else{
            return
        }
        guard let composition_id = dict["id"] as? String else{
            return
        }
        guard let start_position = dict["start_position"] as? [Int] else{
            return
        }
        guard let player_one_dict = dict["player_one"] as? Parameter else{
            return
        }
        guard let player_two_dict = dict["player_one"] as? Parameter else{
            return
        }

        do{
            let player_one = try JSONDecoder().decode(Player.self, from:
player_one_dict.data()!)
            let player_two = try JSONDecoder().decode(Player.self, from:
player_two_dict.data()!)
    let to_vc = CompositionViewController()
            to_vc.type = 2

            //判断哪一个玩家是自己，将自己置于 playerOne
            if player_one.id == user.id{
                CompositionManager.shared.playerJoin(playerOne:  player_one,
playerTwo: player_two)
            }else{
```

```
                CompositionManager.shared.playerJoin(playerOne:    player_two,
playerTwo: player_one)
            }
            //根据返回的 start_position 数据生成随机的棋局
            let composition = Composition(id: composition_id, startChesses:
CompositionUtil.createQue(start_position))
            CompositionManager.shared.compositionCreate(composition:
composition)
            self.present(to_vc, animated: true, completion: nil)
        }catch{

        }
    }
```

接下来我们处理 type 段等于 wait 的情况。这时我们需要在这里监听匹配成功的消息，如何监听呢？我们可以使用代理模式，定义一个协议。

```
    protocol MatchingProtocol: NSObjectProtocol {
            func matchingSucceed(_ dict: [String: Any])
    }
```

给 MessageManager 类里加上这个协议的属性。

```
    weak var matchingDelegate: MatchingProtocol?
```

然后我们在收到消息时进行判断，收到的消息也是有 type 字段的，我们前面也提到过，收到连接成功的消息时，type 字段等于 init。这里当收到匹配成功时，type 字段等于 matched，同时和前面匹配成功后一样，返回相应的棋局数据，只是这里用的是 message 字段。

```
        //处理消息数据
    func handleResponse(_ result: [String: Any]){
        guard let type = result["type"] as? String else{
            return
        }

        if type == "init"{
            self.client_id = result["client_id"] as? String
            self.bindClient()
            return
        }

        guard let messageDict = result["message"] as? [String: Any]  else{
            return
        }
        //匹配成功
        if type == "matched"{
            self.matchingDelegate?.matchingSucceed(messageDict)
            return
```

```
        }

    }
```

接下来首先让 HomeViewController 遵循 MatchingProtocol 协议，然后实现其对应的方法，当监听到匹配成功后，调用联网匹配成功后的开局方法 startOnlineGame。

```
    extension HomeViewController: MatchingProtocol{
            func matchingSucceed(_ dict: [String : Any]) {
            //监听到成功，开局
            self.startOnlineGame(dict)
            }
    }
```

最后就是在 startMatch 方法里加入对不同 type 的处理。

```
          if type == "wait"{
        //匹配中，开始监测匹配成功
        MessageManager.shared.matchingDelegate = self
        return
          }
          //开局
      if type == "start"{
        self.startOnlineGame(result)
        return
      }

    CompositionManager.shared.playerJoin(playerOne:  player_two,  playerTwo:
player_one)
```

之前我们做单机版游戏时，默认棋盘下面的玩家先走棋，但这里是联网匹配，双方看到的界面都是自己在棋盘下面，我们需要处理的是谁先走棋的情况。因此，还需要后台能够告诉我们当前是谁先走棋。

因为我们之前在 CompositionManager 的 playerJoin 方法里直接设置 self.curPlayer = self.playerOne，所以这里就需要进行一定的处理了。我们规定是后台返回的 player_one 字段对应的玩家先走棋，我们直接在 startOnlineGame 方法完成调用 playerJoin 后重新设置当前玩家即可。

```
          //判断哪一个是玩家是自己，将自己置于 playerOne
      if player_one.id == user.id{
          CompositionManager.shared.playerJoin(playerOne:player_one,playerTwo:
player_two)
    }else{
          CompositionManager.shared.playerJoin(playerOne:player_two,playerTwo:
player_ one)
      }
      CompositionManager.shared.curPlayer = player_one
```

这里我们测试时需要同时在两台设备或模拟器中运行项目，然后登录两个不同的账号，进行匹配测试。

5．走棋数据的传输

开局后，我们需要走棋了。走棋数据的传输也是按照之前的方式，通过 http 请求发送走棋数据，通过 websocket 接收对方走棋数据。

我们首先在 CompositionManager 里添加一个将走棋数据提交给服务器的方法。

```
        //上传走棋数据
    func uploadStep(_ step: Step){
      let parameter: Parameter = ["uid": ""]

      NetworkUtil.shared.request(urlStr:APIDefine.Composition.move,parameter:
parameter) { (response) in

      }
    }
```

在 move(_ chess: Chess, direction: StepDirection)方法里加入判断，若当前是自己走棋，则将数据传给服务器。

```
        //当前是联网对战，将自己的走棋数据传给对方
    if self.type == .online{
      if self.curPlayer.id == self.playerOne.id{
        self.uploadStep(step)
      }
    }
```

之后我们需要监听接收对方的走棋数据消息，可以和之前监听匹配结果一样进行处理，首先定义一个走棋协议。

```
    protocol StepProtocol: NSObjectProtocol {
            func didStep(_ dict: [String: Any])
    }
```

接着给 MessageManager 添加这个协议对应的属性。

```
    weak var stepDelegate: StepProtocol?
```

然后在 MessageManager 的 handleResponse 方法的后面加上走棋数据的判断，并调用代理方法。

```
        //走棋数据
      if type == "step"{
    self.stepDelegate?.didStep(messageDict)
        return
      }
```

接下来就是在 CompositionManager 里实现这个协议了，我们可以在其 type 属性改变后添加相关的代理。

```
        var type: CompositionType = .outline{
    didSet{
        if type == .online{
          MessageManager.shared.stepDelegate = self
        }
    }
          }
```

最后让 CompositionManager 遵循这个协议，并实现对应的方法。

```
    extension CompositionManager: StepProtocol{
            func didStep(_ dict: [String: Any]){
            let step = Step(count: 0, playerId: "", chess: Chess(belong: 1, value:
2), direction: .down )
            self.move(step)
            }
    }
```

这里基本都是在 CompositionManager 进行走棋业务处理的，那么视图界面需要怎么处理呢？其实我们和之前的 AI 走棋一样，前题就是当前玩家将棋盘置为可操作即可。在控制器 CompositionViewController 的 playerSwitched 方法里加入对应的判断。

```
          if manager.type == .online {
        if manager.curPlayer.id == manager.playerTwo.id{
          self.compositionView.isUserInteractionEnabled = false
        }else{
          self.compositionView.isUserInteractionEnabled = true
        }
    }
```

6．联网对战的游戏结束

联网对战游戏结束时，和之前的处理也是稍有不同的，单机版时提示谁获胜，联网对战需要提示当前用户是胜了还是败了。这里我们只要在 CompositionViewController 的 gameOver 方法里稍加修改即可。

```
        func gameOver(_ manager: CompositionManager) {

        let curName = manager.curPlayer.name
        var title =  "游戏结束"
        var message = "\(curName)获胜啦"
        if manager.type == .online{
            if manager.curPlayer.id == manager.playerOne.id{
                //自己获胜
                title = "胜"
                message = "您获胜了"
            }else{
```

```
                title = "败"
                message = "您失败了"
            }
        }
        let alertView = UIAlertController(title: title, message: message,
preferredStyle: .alert)
        let actionNext =UIAlertAction(title:"下一局",style:.default){(action)in
            self.newGame()
        }
        let actionBack = UIAlertAction(title:"返回",style:.default) { (action)in
            self.back()
        }
        alertView.addAction(actionNext)
        alertView.addAction(actionBack)
        self.present(alertView, animated: true, completion: nil)
    }
```

7．发起悔棋请求

由于是联网对战，所以玩家发起悔棋请求时，需对方同意才可以悔棋，这里自然就需要一个发起悔棋的 http 请求了，也需要监听 websocket 发来对方申请悔棋的消息，以及监听对方是否同意悔棋的消息。

在 APIDefine 的 Composition 下定义申请悔棋接口，以及是否同意悔棋接口。

```
    //悔棋请求
    /*
    * 用户 id：uid
    */
    static let applyRegret = "\(BasePath)/composition/apply_regret"

    //是否同意悔棋
    /*
    * 用户 id：uid
        * 是否同意：agree，1：同意，2 拒绝
    */
    static let regretResult = "\(BasePath)/composition/regret_result"
```

在 CompositionManager 里定义申请悔棋的方法，并调用 http 接口。

```
      //申请悔棋
    func applyRegret(){
        //如果不是联网匹配，则直接执行悔棋功能
        if self.type != .online{
            self.regret()
            return
```

```
        }

        let parameter: Parameter = ["uid": ""]

        NetworkUtil.shared.request(urlStr: APIDefine.Composition.applyRegret,
parameter: parameter) { (response) in

        }

    }
```

将 CompositionViewController 里申请悔棋时调用的方法 regret()改为

```
    self.compositionManager.applyRegret()
```

在 StepProtocol 协议里添加申请悔棋和监听悔棋结果的消息。

```
    protocol StepProtocol: NSObjectProtocol {
            func didStep(_ dict: [String: Any])
            //对方申请悔棋
            func applyRegret(_ messageManager: MessageManager)
            //对方处理悔棋申请结果
            func resultRegret(_ messageManager: MessageManager, agree: Bool)
    }
```

在 MessageManager 里 handleResponse 监听对方消息，处理相应的 type。

```
          //申请悔棋
        if type == "applyRegret"{
            self.stepDelegate?.applyRegret(self)
            return
        }

        //同意悔棋
        if type == "agreeRegret"{
            self.stepDelegate?.resultRegret(self, agree: true)
            return
        }

        //拒绝悔棋
        if type == "refuseRegret"{
            self.stepDelegate?.resultRegret(self, agree: false)
            return
        }
```

我们在 CompositionManager 里监听消息结果，但最终还需要将结果传给控制器，所以在 CompositionManagerDelegate 协议里也加上对应的方法。

```
          //申请了悔棋
```

```
    func applyRegret(_ manager: CompositionManager)

    //悔棋结果
    func resultRegret(_ manager: CompositionManager, agree: Bool)
```

在 CompositionManager 里实现 StepProtocol 协议的方法，并通过代理传给控制器。

```
        //对方申请悔棋
        func applyRegret(_ messageManager: MessageManager){
    self.delegate?.applyRegret(self)
        }
        //对方悔棋处理结果
        func resultRegret(_ messageManager: MessageManager, agree: Bool){
    self.delegate?.resultRegret(self, agree: agree)
        }
```

在 CompositionManager 里定义是否同意悔棋结果的方法，将是否同意悔棋结果通过 http 请求传给服务器。

```
        //是否同意悔棋
    func agreeRegret(_ agree: Int){
        let parameter: Parameter = ["uid": "",
                      "agree": agree]

        NetworkUtil.shared.request(urlStr: APIDefine.Composition.applyRegret,
parameter: parameter) { (response) in

        }

    }
```

最后在 CompositionViewController 控制器里实现 CompositionManagerDelegate 的方法。

```
        //申请了悔棋
    func applyRegret(_ manager: CompositionManager){
        let alertView = UIAlertController(title: "提示", message: "对方申请悔棋，
是否同意", preferredStyle: .alert)
        let actionAgree=UIAlertAction(title:"同意",style:.default) { (action)in
            self.compositionManager.agreeRegret(1)
        }
        let actionRefuse=UIAlertAction(title:"拒绝",style:.default){(action) in
            self.compositionManager.agreeRegret(0)
        }
        alertView.addAction(actionAgree)
        alertView.addAction(actionRefuse)
        self.present(alertView, animated: true, completion: nil)
    }
```

```
    //悔棋结果
    func resultRegret(_ manager: CompositionManager, agree: Bool){
        if agree{
            self.compositionManager.regret()
        }else{
            let alertView = UIAlertController(title: "提示", message: "对方不同
意悔棋", preferredStyle: .alert)
            let actionSure = UIAlertAction(title: "确定", style: .cancel, handler:
nil)
            alertView.addAction(actionSure)
            self.present(alertView, animated: true, completion: nil)
        }
    }
```

8．玩家认输

玩家中途认输，只需认输方通过 http 请求将认输信息告诉服务器，然后服务器通过 wezsocket 将认输信息告诉另一方。整个流程和前面申请悔棋的流程一样。

定义接口。

```
        //认输接口
    /*
    * 用户 id：uid
    */
    static let giveUp = "\(BasePath)/composition/giveUp"
```

在 StepProtocol 协议里加入新方法。

```
        //对方认输
    func giveUp(_ messageManager: MessageManager)
```

在 MessageManager 的 handleResponse 方法里加入对消息认输的判断。

```
        //认输
    if type == "giveUp"{
        self.stepDelegate?.giveUp(self)
        return
    }
```

在 MessageManager 的 handleResponse 方法里加入对消息认输的判断。

```
        //认输
    if type == "giveUp"{
        self.stepDelegate?.giveUp(self)
        return
    }
```

在 CompositionManager 里定义认输方法。

```
        //发送认输
    func giveUp(){
        let parameter: Parameter = ["uid": ""]

        NetworkUtil.shared.request(urlStr:      APIDefine.Composition.giveUp,
parameter: parameter) { (response) in

        }

      }
```

在 CompositionManagerDelegate 协议里加入方法。

```
        //认输
    func giveUp(_ manager: CompositionManager)
```

在 CompositionManager 里实现 StepProtocol 协议的认输方法。

```
        func giveUp(_ messageManager: MessageManager) {
        self.delegate?.giveUp(self)
    }
```

在 CompositionViewController 里单击认输按钮的事件添加判断是否为联网对战。

```
        @objc func didSurrenderButton(){
        let alertView = UIAlertController(title: "提示", message: "确认认输",
preferredStyle: .alert)
        let actionNext = UIAlertAction(title: "重新游戏", style: .default)
{ (action) in
           if self.compositionManager.type == .online{
              self.compositionManager.giveUp()
           }
           self.newGame()
        }
        let actionOut = UIAlertAction(title: "退出游戏", style: .default)
{ (action) in
           self.back()
        }
        let actionCancel = UIAlertAction(title: "取消", style: .cancel, handler:
nil)
        alertView.addAction(actionNext)
        alertView.addAction(actionOut)
        alertView.addAction(actionCancel)
        self.present(alertView, animated: true, completion: nil)
    }
```

在 CompositionViewController 里实现 CompositionManagerDelegate 协议的相关方法。

```
        func giveUp(_ manager: CompositionManager) {
```

```
      let alertView = UIAlertController(title: "提示", message: "游戏结束，对方
认输", preferredStyle: .alert)
      let actionSure = UIAlertAction(title:"确定",style:.default){(action)in
        self.dismiss(animated: true, completion: nil)
      }
      alertView.addAction(actionSure)
      self.present(alertView, animated: true, completion: nil)
    }
```

至此，完成了该案例小游戏的基本业务功能，当然还有些需要优化的地方。

9．玩家头像的加载显示

这里我们使用第三方的图片请求缓存库 SDWebImage 加载网络头像。首先在 Podfile 文件里加入导入该库的指令。

```
    target 'AnimalCardChess' do
           pod 'SnapKit', '~> 4.0.0'
           pod 'Starscream'
           pod 'SDWebImage'
    end
```

然后用终端输入 pod install，完成 SDWebImage 库的导入，接下来就是对第三方库的使用了。首先我们在 Vender 文件夹下新建一个目录 SDWebImage，然后创建一个 ImageCacheExtension.swift 文件，如图 24-33 所示。

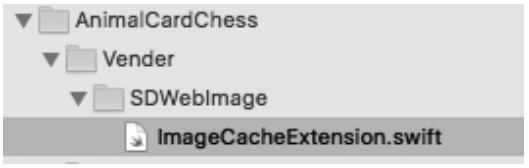


图 24-33

这么做是为了对第三方库方法进一步封装，而不是直接使用，其目的是为了防止后期若觉得第三方库不好用，想更换第三方库，或者可以对第三方库的方法进行改进。

```
    import UIKit
    import SDWebImage

    extension UIImageView{
           func mc_setImage(with url: URL!, placeholderImage placeholder:
UIImage! = nil){
           self.sd_setImage(with: url, placeholderImage: placeholder)
           }
    }
```

然后对 PlayerView 里的 update 方法进行修改。

```
         if CompositionManager.shared.type == .online{
```

```
        self.avatarImageView.mc_setImage(with:URL(string: self.player!.avatar))
    }else{
        self.avatarImageView.image = UIImage(named: self.player!.avatar)
    }
```

24.8 联网对战的优化

我们已经完成了该案例的基本功能，下面再对联网对战过程中的一些地方进行优化。

1. 登录页加载框的封装

前面我们虽然实现了登录功能，但登录过程因为是一个网络请求的过程，所以这其中自然就会有一定的等待时间，而且没有处理登录失败的提示。对此，我们可以封装一个加载提示框。

我们在 Common 的 Customer 路径下新建一个文件夹 MCLoading，然后我们创建新的文件 MCLoadingView 类，并继承于 UIView，如图 24-34 所示。

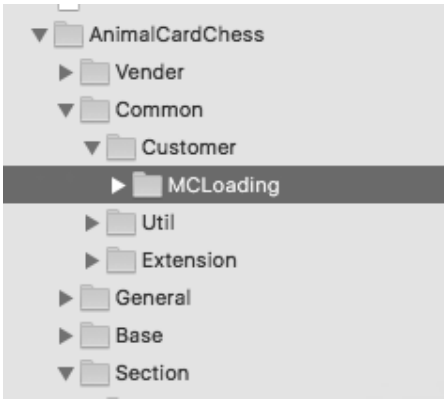


图 24-34

```
        import UIKit

    class MCLoadingView: UIView {

        lazy var activityViewFor = UIActivityIndicatorView(activityIndicator
Style: .whiteLarge)
        lazy var activityBackView = UIView()
        lazy var loadingLabel = UILabel()

        var loadingBgColor: UIColor = UIColor.black.withAlphaComponent(0.8)
```

```
override init(frame: CGRect) {
    super.init(frame: frame)
    self.createView()
    self.createSnp()
}
required init?(coder aDecoder: NSCoder) {
    fatalError("init(coder:) has not been implemented")
}

convenience init(){
    self.init(frame: CGRect.zero)
}

func createView(){
    self.activityBackView.backgroundColor = self.loadingBgColor
    self.activityBackView.layer.cornerRadius = 5
    self.activityBackView.layer.masksToBounds = true
    self.addSubview(self.activityBackView)

    self.activityBackView.addSubview(self.activityViewFor)

    self.loadingLabel.textColor = UIColor.white
    self.loadingLabel.textAlignment = NSTextAlignment.center
    self.loadingLabel.font = UIFont.systemFont(ofSize: 12)
    self.activityBackView.addSubview(self.loadingLabel)
}

//加载中的布局
func createSnp(){
    self.activityBackView.snp.remakeConstraints { (make) in
        make.width.height.equalTo(120)
        make.center.equalToSuperview()
    }
    self.activityViewFor.snp.remakeConstraints { (make) in
        make.width.height.equalTo(60)
        make.centerX.equalToSuperview()
        make.centerY.equalToSuperview().offset(-10)
    }
    self.loadingLabel.snp.remakeConstraints { (make) in
        make.height.equalTo(20)
        make.left.right.equalToSuperview().inset(10)
        make.top.equalTo(self.activityViewFor.snp.bottom)
```

```
        }
    }

    //只有提示时的布局
    func updateSnp(){
        self.loadingLabel.textAlignment = .center
        self.activityBackView.snp.remakeConstraints { (make) in
            make.width.equalTo(180)
            make.center.equalToSuperview()
        }

        self.activityViewFor.snp.remakeConstraints { (make) in

        }

        self.loadingLabel.snp.remakeConstraints { (make) in
            make.left.right.equalToSuperview().inset(5)
      make.top.bottom.equalToSuperview().inset(10)
        }
    }

    //停止之前的动画效果
    func removeAnimation(){
        self.layer.removeAllAnimations()
        self.activityBackView.layer.removeAllAnimations()
        self.activityViewFor.layer.removeAllAnimations()
    }


    //开始加载
    func startLoading(_ text: String){
        self.removeAnimation()
        self.createSnp()
        self.alpha = 1
        self.loadingLabel.text = text
        self.activityViewFor.startAnimating()
    }

    //结束加载
    func stopLoading(_ alertString: String = ""){
        if alertString != ""{
            self.removeAnimation()
```

```
                self.activityViewFor.stopAnimating()
                self.loadingLabel.text = alertString
                self.updateSnp()
            }
            self.dismiss()
        }

        //提示
        func alert(_ text: String){
            self.removeAnimation()

            self.addToWindows()

            self.snp.remakeConstraints { (make) in
                make.edges.equalToSuperview()
            }

            self.alpha = 1
            self.loadingLabel.text = text
            self.updateSnp()
            self.dismiss()
        }

        //消失动画
        func dismiss(){
            UIView.animate(withDuration: 0.35, delay: 1.5, options: UIViewAnima
tionOptions.allowUserInteraction, animations: { () -> Void in
                self.alpha = 0
            }) { (f) -> Void in
                if f{
                    self.activityViewFor.stopAnimating()
                    self.removeFromSuperview()
                }
            }
        }

        func addToWindows(){
            self.frame = UIScreen.main.bounds
            if let window = UIApplication.shared.keyWindow{
                window.addSubview(self)
                return
            }
```

```
        }

    }
```

这里我们通过 UIView 和系统自带的菊花旋转控件 UIActivityIndicatorView，并添加一些出现和消失的动画效果，完成了一个简单的加载提示框。

我们还需要对这个 UIView 控件进一步封装，定义一个单例，以方便在其他地方可以直接调用。继续在 MCLoading 文件夹下新建类文件 MCLoading。

```
        import UIKit

    class MCLoading: NSObject {
       static let shared = MCLoading()

        lazy var loadingView = MCLoadingView()

        var target: UIViewController!
        func startLoading(_ target: UIViewController! = nil, text: String =
"努力加载中..."){
            self.loadingView.startLoading(text)

            if let _ = target{
                self.target = target
                target.view.isUserInteractionEnabled = false
                target.view.addSubview(self.loadingView)
            }else{
                self.loadingView.addToWindows()
            }
            self.loadingView.snp.remakeConstraints { (make) in
                make.edges.equalToSuperview()
            }
        }

        func stopLoading(_ alertString: String! = nil){
            DispatchQueue.main.async {
                self.target?.view.isUserInteractionEnabled = true
                if alertString != nil{
                    self.loadingView.stopLoading(alertString)
                    return
                }
                self.loadingView.stopLoading()
            }
        }
```

```
    func alert(_ text: String){
        DispatchQueue.main.async {
            self.loadingView.alert(text)
        }
    }
}
```

完成之后，我们就可以在登录界面进行调用了，对 LoginViewController 控制器中的 didLoginButton 方法进行修改。

```
    @objc func didLoginButton(){
        guard let account = self.accountTextField.text, account != "" else{
            MCLoading.shared.alert("请先填写账号")
            return
        }
        guard let password = self.pwdTextField.text, password != "" else{
            MCLoading.shared.alert("请先填写密码")
            return
        }
        MCLoading.shared.startLoading(text: "登录中...")
        UserManager.shared.login(account,password:password){(errorMsg)in
            if errorMsg == nil{
                //登录成功
                MCLoading.shared.stopLoading("登录成功")
                self.dismiss(animated: true, completion: nil)
            }else{
                //登录失败
                MCLoading.shared.stopLoading("登录失败")
            }
        }
    }
```

2．添加取消登录按钮

如果登录失败，我们应可以取消登录，从登录界面返回首页。先在 LoginViewController 控制器定义一个按钮。

```
    lazy var cancelButton: UIButton = {
let btn = UIButton()
btn.setTitle("取消", for: .normal)
btn.setTitleColor(Color.textH, for: .normal)
btn.titleLabel?.font = Font.middle
btn.addTarget(self,action:#selector(self.cancelLogin),for:.touchUpInside)
return btn
}()
```

在 createView 方法里将 cancelButton 添加到主视图上。

```
self.view.addSubview(self.cancelButton)
```

最后实现单击取消按钮的方法。

```
@objc func cancelLogin(){
    self.dismiss(animated: true, completion: nil)
}
```

案例我们就暂且介绍到此了。联网对战部分限于篇幅，我们只对功能进行了简单的实现。整个案例编写可能并不完美，但希望能够起到抛砖引玉的作用，给读者在进行实战项目开发时一点参考和启发。